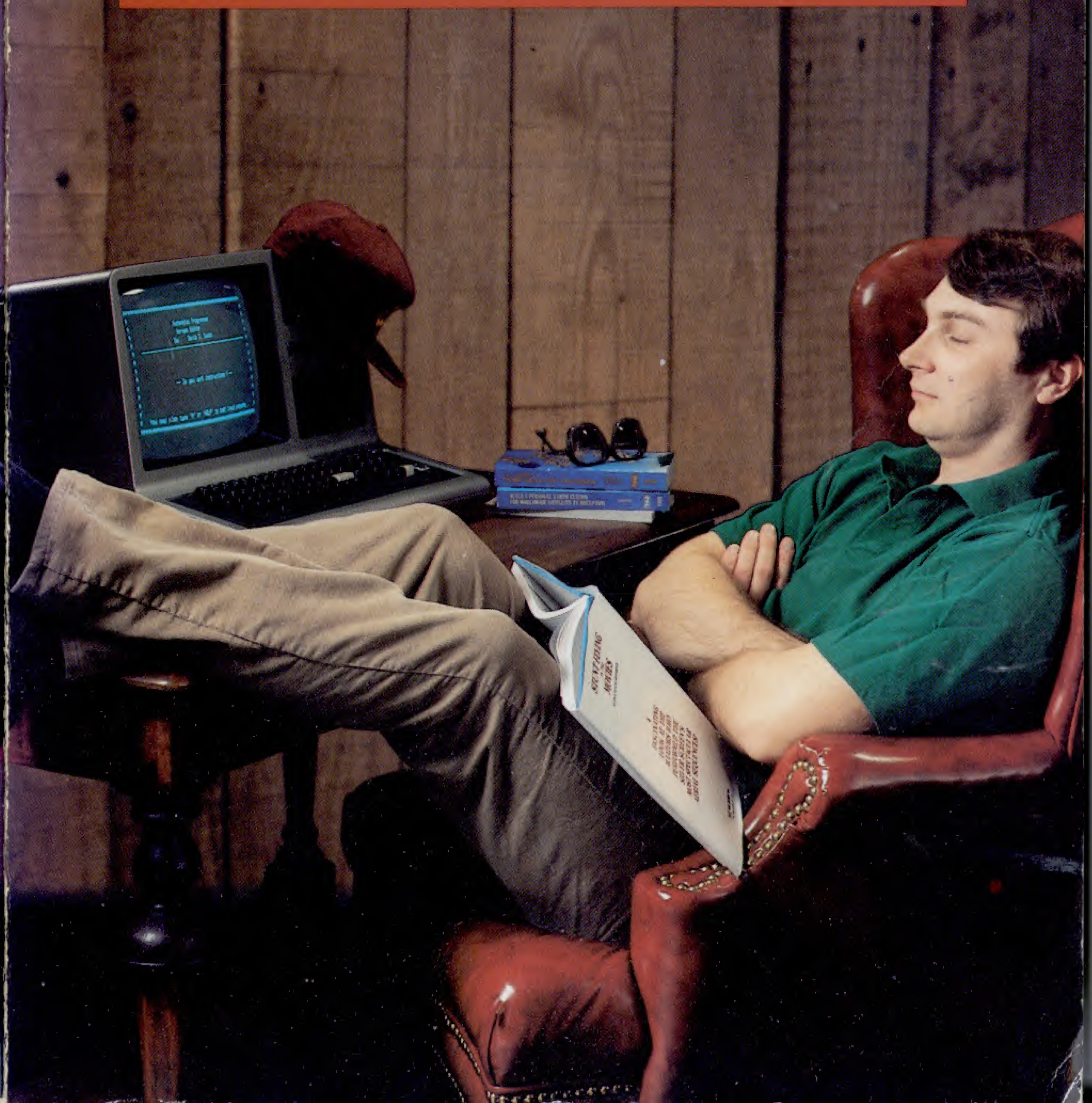
1798

TEACH YOUR TRS-80™ TO PROGRAM ITSELF!

BY DAVID BUSCH



TEACH YOUR TRS-80™ TO PROGRAM ITSELF

BY DAVID BUSCH

TAB BOOKS Inc.
BLUE RIDGE SUMMIT, PA. 17214

Also by the Author from TAB Books Inc.

No. 1698 *25 Games for Your TRS-80 Model 100*

FIRST EDITION

FIRST PRINTING

Printed in the United States of America

Library of Congress Cataloging in Publication Data

Busch, David D.
Teach your TRS-80 to program itself!

Includes index.
1. TRS-80 Model I (Computer)—Programming. 2. TRS-80 Model III (Computer)—Programming. 3. TRS-80 Model 4 (Computer)—Programming. I. Title. II. Title: Teach your T.R.S.-80 to program itself! III. Title: Teach your TRS-eighty to program itself!
QA76.8.T1833B87 1984 001.64'2 84-2525
ISBN 0-8306-0798-6
ISBN 0-8306-1798-1 (pbk.)

Contents

Acknowledgment

The material in Chapters 4, 6, 7, 8, and 16 originally appeared in different form in *Interface Age Magazine.*

Introduction

Yes, your TRS-80 I/III or 4 *can* write its own programs. Instead of laboriously crafting program lines and subroutines that will display a series of instructional "frames" on the screen, you can let your computer do all that work. You need only design the screen, using word processing commands, and tell the computer how long you want that frame displayed. The TRS-80 is perfectly capable of writing a complete program that will do exactly that, without the need for you to write one single line of code.

Or your TRS-80 can compose subroutines for you. Do you need some disk input/output routines and a string array to store data in? Some data lines, perhaps? A menu? But you're not too eager to write the code and figure out the proper ON . . . GOTO lines? That task is a snap for the automatic TRS-80.

On the other hand, you may be weary of calculating tabs for neatly formatted screen displays. Wouldn't it be nice to just type PRINTTAB(T) and let the computer figure out what value T should be? Say no more. Your wish is well within the capabilities of the Fort Worth Wonder.

As fabulous a set of tools as the TRS-80 line has been, most users save only half the time they could with their computers. Because I write dozens of programs a year, one of the first things I did was write a number of programs that do nothing more than write other programs for me. One of the first, and one I use more than any other, was Screen Editor. It is simply a BASIC program that allows

drawing on the screen any menu, title block, instructional screen, or other material that will be needed in a program. Then, at the press of the ENTER key, the screen just designed is magically transformed into program lines. Ten minutes of coding can be accomplished in a minute or less. (Since I have compiled into machine code the BASIC Screen Editor shown in this book, the chore takes no more than a second or two!)

Given the right tools, such as the 17 utility programs here, an hour spent programming can be more fruitful than several hours with manual methods. A third (or more) of the program lines in some of the examples in this book were prepared by other programs listed. Some programs were even used to write enhanced versions of themselves.

All the programs in this book will work with TRS-80 Model I, Model III, or Model 4 computers, and were written and tested on all three. Some will run equally well with Model 4s as Model IIIs. Tabber, for example, asks the user if tabs should be centered for a 64-column screen, or an 80-column screen. Because they are written in BASIC and use few PEEKs, the programs are readily transferable.

Those which PEEK into video memory, however, work on the Model 4 only when the computer is operated in Model III mode. Since the Model 4 has not been long on the market, and software is still sparse, most users will have a Model III operating system, such as TRSDOS 1.3, NEWDOS/80 2.0, or LDOS 5.1 to use Model III applications programs on their Model 4. Those users, then, can run some of the programs in this book in Model III mode, create the programs of their choice, and then transfer them to Model 4 formatted disks for running. A few changes may have to be made to account for the 80-column screen once the programs have been transferred. In fact, Tabber, a program included in this book, can do part of this chore for you.

Just as PEEKs and POKEs have been avoided wherever possible, other statements that are "DOS dependent" have been avoided. In most cases, strictly BASIC syntax common to all TRS-80 computers is used. That is, all Radio Shack Model I/III/4 computers have similar disk input/output routines for sequential files. Exotic file types are not used. In a few cases, special features of popular disk operating systems are incorporated as options. Program Proofer, for example, asks if the user is running NEWDOS/80. If so, the program will use CMD"O" to sort the variable and word list. Menu Master calls up various DOS com-

mands using the 'CMD"n" ' syntax. Since users will customize this program anyway, those with TRSDOS 6.0 or LDOS can change the CMD to SYSTEM, and make other minor changes.

This book is only a jumping-off point. Many of the programs were adapted from other programs. Visual Maker is based on Screen Editor. Global Replacer is descended from Tabber. Similarly, you can take ideas and suggestions here and develop programs of your own that will streamline your BASIC development work. In addition, there are some ideas in Chapter 18 for using as shortcuts other programs you already own, such as word processors or keyboard utilities.

The 17 utility programs in "The Automatic TRS-80" actually write programs for you, modify existing software, or give your programs new capabilities and power. The novice or experienced programmer can save hours of time on every program written. Some of the examples were used to write programs in this book, or to modify themselves. Here is a brief outline of the programs:

Visual Maker. Design a custom "slide" to appear on the screen of your TRS-80, using graphics or alphanumeric characters. Tell Visual Maker how long you want that slide to be displayed. Then go onto the next slide. Or, you can draw from a library of slides you have compiled.

Once your show is assembled into the order you want, Visual Maker will write a complete BASIC program to display the slides you designed for the intervals requested. *No* programming is required. You can even repeat the shows if you like, or end with a menu that allows the user a choice of which show to see next.

DB Starter. Weary of writing custom database management programs from scratch? DB Starter will do the basic skeleton for you. Enter the number of menu choices, and the prompts to be included in the menus. It will design the menu for you. Tell the program you want input/output routines, feed in a few parameters, and it will write the I/O modules automatically. DB Starter will also construct the necessary ON...GOTO lines, and insert REMarks at line numbers where the programmer needs to build up the BASIC skeleton. Your first several hours of programming are taken care of for you.

Tabber. Want to center your screen output for prompts and other messages? Just type PRINTTAB(T) in every line you want centered. This program will go through an entire program, calculate how long the message is, and write a new program line that TABs the proper number of spaces. It works with 80-column Model 4 screens, as well as 64-column Model I and III screens.

Proofer. Find misspelled keywords, mismatched parentheses, and other errors *before* runtime. This program helps you debug, and provides a list of variable names used in the program as a bonus.

Error Message. If you are impressed with the long error messages of Disk BASIC, this program will knock you flat. Append Error Message to your own BASIC program and insert the appropriate ON ERROR GOTO line. Any error will then be spelled out in detail—with tips on how to find the exact error in your program. It is an excellent utility for novice programmers, or anyone tracking an elusive bug.

Screen Editor. You can use word processing command style to design a custom screen. Then, press ENTER. This program writes the BASIC program lines you need to reproduce your custom screen in your own program.

Documenter. Type out the instructions to go with your latest program. Documenter will divide them up and write BASIC program lines that present them as instructions at the beginning of your program. It includes everything, even "Hit any key for next page" prompts.

Menu Master. This program will let your computer AUTO on power-up into a custom menu that summons the programs you use most often, gives you directories of your disk drives, and does a number of other small but useful jobs.

Word Counter. Why write a program to count words in files, when you could do it manually almost as fast? Or could you? Suppose you were writing a 10,000-word term paper, or a 70,000-word book which had to come out at a contracted number of printed pages. Word Counter will be far more useful to you than you might think.

Global Replacer. Specify a string in your program of file—it does not have to be a keyword—and this program will replace it with the string of your choice.

REM-over. When you're down to your last blank disk or your last few bytes of memory at 3 o'clock in the morning, you'll appreciate the value of stripping the REM statements from a program. REM-over does it for you in seconds.

Lister. Do you envy those glitzy-looking listings but hate the work involved in formatting them, and begrudge the memory those embedded spaces kill? Lister will help you out.

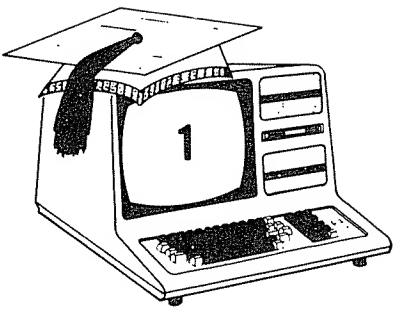
Translator. This program creates a Spanish-language tiny BASIC to use as an educational aid, and then "compiles" the code

created into runnable BASIC. The methodology used in Translator also shows you how to write your own tiny BASICs in French or the language of your choice.

Chain Zapper. If you're a NEWDOS/80 user, do you dread the mailman bearing envelopes full of ZAPs for your operating system? Chain Zapper writes chain files automatically to apply the patches to your DOS.

Titler. Create title listings for your programs simply by entering the title itself. Titler will do the formatting, add your name and address, and write the program lines for you.

The goal of this book is to help you to remember, when you're slaving over a hot keyboard at 3 A.M., that computers are the servants of mankind—and not vice versa.



Word Counter

Why not let your TRS-80 write its own programs? After all, much program writing is nothing more complicated than building something from an inventory of prefabricated subroutines. Many programs have a great deal in common; it is the parameters which change. Wouldn't it be simpler just to provide the parameters—and let the computer do the routine stuff?

After all, one program may require a line like, FOR N=1 TO 100, while the next will need FOR N=1 TO 200. Yet, each time, the programmer had to type in the "FOR N=1 TO" part. The reason the computer never knew enough to supply the "FOR N=1 TO" is that nobody *told* it to. The TRS-80 Model I/III and 4 computers can do almost anything in the area of program-writing, if they are only told exactly what to do.

"Applications generators" and other programs which write other programs are old hat. They have been around for a number of years, and can be purchased for large computers as well as small. The concept behind them is simple: many programs have modules that are much like those used in other software. Yet, in many cases, the computer programmer writes a routine from scratch each time it is needed. Why not build a library of routines and let the computer draw on them as needed to write its own programs?

The reason a TRS-80 Model I/III/4 can write its own BASIC programs lies in its ability to load two types of files into BASIC from disk. The normal way a BASIC program is saved is in compressed

format. That is, BASIC keywords are tokenized, and, instead of the entire keyword, a single byte representing that keyword is loaded onto the disk. Rather than store the five letters that make up "PRINT", BASIC normally just stores 178, the decimal number that represents "PRINT". When you type SAVE"filename", a program is stored on disk in this form.

However, we can also type SAVE"filename",A. Then, the program will be saved in noncompressed "ASCII" format. That is, every letter and number will be stored byte for byte on the disk, exactly as the program appears when listed. The BASIC interpreter has the capability of doing this conversion for us.

An ASCII file is nothing more than a text file. It is possible to load a noncompressed program into a word processing program such as Scripsit, edit it using powerful global search and replace commands, and then save it back to disk in ASCII form.

Because of this dual capability we can also create programs using a word processor, or, in the case of the programs in this book, through the use of sequential disk files, which are also ASCII files. The short program below serves as an example:

```
10 OPEN "O",1,"TEST"
20 PRINT#1,"10 PRINT";CHR$(34);"THIS IS A
   TEST";"CHR$(34)
30 CLOSE 1
```

That test program will write a single line to the disk under the filename "TEST." That line will be, if loaded from BASIC, a short program in the form:

```
10 PRINT"THIS IS A TEST"
```

We could also "build" the program lines from our own parameters. Try this short program:

```
10 INPUT"Enter line number desired:";LN
20 INPUT"Enter message desired :";MESS$
30 INPUT"Want it to be PRINT or LPRINT";CH$
40 IF CH$="PRINT" OR CH$="LPRINT" GOTO 60
50 GOTO 30
60 OPEN "0",1,"TEST"
70 PROG$=STR$(LN)+CHR$(32)+CH$+CHR$(32)+
   CHR$(34)+ MESS$+CHR$(34)
80 PRINT#1,PROG$
90 CLOSE 1
```

Most of the programs in this book with program-writing routines do nothing more than assemble program lines in this manner. Sometimes the input comes from the user. Other times it is calculated. Still other times, some of the programs PEEK the video memory (from 15360 to 16384) to see what has been printed to the screen, and use that.

But the common thread is the use of ASCII files, which are programs, as if they were data files. The first program presented, "Word Counter," illustrates the principle, though it does not create any new program files itself. Instead, Word Counter reads in an ASCII file, and counts the number of words. Most commonly these files will be word processing text files, like those created by Scripsit. However, Word Counter could just as easily be used to count the number of words in a program.

Most of the techniques used in this book will be repeated in later programs. Each will be explained in detail the first time it is used. So, early programs are short because explanations are frequent; later, longer programs will use many techniques that have been previously explained and will thus require fewer discussions.

Programs in this book frequently access other programs that have been stored in ASCII form on disk. You *must* save a program to be used by another program in ASCII form, using the ",A" option. If, in running one of the programs here, you see garbage on the screen, you probably forgot to save the program in ASCII.

Word Counter is no exception. It will count words in a program file the same as a text file, but only if both are in ASCII. The operator is asked to enter the name of the file to be processed in line 200. That file, F$, is opened, and one line is input from the disk. The line is loaded by means of LINEINPUT#1 in line 220. INPUT#1will accomplish much the same thing, except that it will not

A$	Stores text line being examined.
AW	Average word length in text.
C$	One-character string from middle of line.
CHAR	Number of characters in whole file.
CU	Counter of number of words in file.
F$	Text file to be counted.
FL	FLAG indicating end of file reached.
L$	Last character encountered.
N	Loop counter.
SW	Number of standard words in text.

Fig. 1-1. Variables used in Word Counter.

```
10 ' **********************
20 ' *                    *
30 ' *    Word Counter    *
40 ' *                    *
50 ' **********************

60 CLEAR 4000
70 DEFINT A-Z

75 ' *** Instructions ***

80 CLS:PRINT:PRINT
90 PRINT TAB(21)"Writer's Word Counter "
100 PRINT
110 PRINT TAB(6)"This program will count the number of actual
   words in a "
120 PRINT TAB(2)"text file, or any file that has been stored to
   disk in ASCII "
130 PRINT TAB(2)"format.   In addition, it also provides the
   total number of "
140 PRINT TAB(2)"'standard ' five-character words, and the
   average character "
```

```
150 PRINT TAB(2)"length of the words in the text. "
160 PRINT:PRINT TAB(17)"== Hit any key to continue == "
170 IF INKEY$="" GOTO 170
180 CLS:PRINT:PRINT
185 ' *** Access Disk File ***
190 PRINTTAB(15)"Enter name of file to count:";
200 LINEINPUT F$
210 OPEN "I",1,F$
220 LINEINPUT #1,A$
225 ' *** If End of File Found, Set Flag to 1 ***
230 IF EOF(1) THEN FL=1
235 ' *** Add Length of A$ to Total Characters in File ***
240 CHAR=CHAR+LEN(A$)
245 ' *** Loop to look at each character in A$ ***
250 FOR N=1 TO LEN(A$)
```

Fig. 1-2. Program listing for Word Counter.

```
260 :       C$=MID$(A$,N,1)
270 :       IF C$=CHR$(32) AND L$<>CHR$(32) THEN CU=CU+1
280 :       L$=C$
290 NEXT N
300 IF FL=1 GOTO 320
310 GOTO 220
315 ' *** Print out Results ***
320 CLS:PRINT:PRINT
330 PRINTTAB(23)"NUMBER OF WORDS =",CU
340 PRINT
350 AW=CHAR/CU
360 PRINTTAB(21)"AVERAGE WORD LENGTH =",AW
370 PRINT
380 SW=CHAR/5
390 PRINTTAB(17)"NO. OF FIVE-CHARACTER WORDS =",SW
400 CLOSE
405 ' *** Run again? ***
410 PRINTTAB(4)"Check another file?"
420 PRINTTAB(10)"Y/N"
430 A$=INKEY$:IF A$=""GOTO 430
440 IF A$="Y" OR A$="y" THEN RUN ELSE CLS
```

Fig. 1-2. Program listing for Word Counter. (Continued from page 5.)

accept string delimiters such as commas and quotation marks, which are commonly used in both text and program lines. LINEINPUT imposes no such restriction. It accepts everything up to the next carriage return. This will be the end of a program line, or a carriage return in the text itself.

To search for a word, we need to first decide just what a word is. The easiest thing is to consider a word to be more or less, a group of letters preceded and followed by a space. "CODEWORD" is one word, even though two real words are embedded in it. "OH! NO!" is two words. The punctuation is not part of each word, but, for the purposes of this program it is considered so. This is because Word Counter has been written to look for each space that is preceded by a non-space. Counting spaces would be an inaccurate way of counting words; the program instead looks at each character, and, when it finds a space, looks to see if the preceding character was a space. If not, the end of a word has been deemed to have been reached.

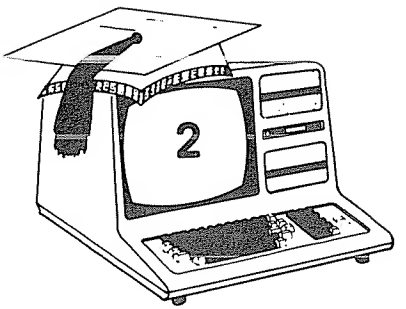
Referring to the variable chart and program listing (Figs. 1-1 and 1-2), we see that each line input, stored in A$, is examined one character at a time in a FOR-NEXT loop beginning at line 250. The loop repeats from 1 to the length of A$. Each time through, C$ is assigned the value of the next character in the string through the use of MID$(A$,N,1).

Before the loop goes back to look at the next character, the current character is stored in L$ (line 280), and becomes the last character.

If C$ is a space (CHR$(32)), the program looks at the last character checked, L$, to see if it was a space. If it was *not* a space (that is, it was a character) then the program assumes the end of a word, since no word contains an embedded space. Thus the word counter CU, is incremented by one.

Once the program has looked at every character in the string, it drops down to line 300, where the end-of-file flag is tested. If it is equal to one, meaning the EOF marker has been reached, the program goes to line 320 to present the results of the word count. Otherwise, the program goes back to line 220 to input another line.

When the file is finished, the program prints the number of words, CU, and then calculates the average word length, which is the number of characters (CHAR) divided by the number of words. The number of characters is also divided by five to total the amount of "standard," five-character words as well.



REM-over

In Chapter 1 we explored opening an ASCII disk file, either text or program, reading it in line by line, and then examining the string of characters in order to count the number of words. The next step is to alter the file in some way, and then write a new, changed file to disk. Several of the programs in this book are based on that principle, and the first of these is "REM-over."

REM-over will read in a disk file, such as that shown in Fig. 2-1, and will print to disk a new file that is similar to the old one. The only difference is that when the program encounters a remark, designated either by "REM" or its apostrophic abbreviation ('), the remainder of the program line will be truncated. If a line consists only of a line number and a remark, the line will be deleted from the program entirely. The result will be a new program with all of the comments removed, as shown in Fig. 2-2. Depending on the number of remarks included in the original program, the new, remarkless version may be significantly smaller, and therefore consume less memory space.

Ordinarily, one might think that removing remarks from a program would be ridiculously simple. Since the TRS-80 ignores anything after REM or,' a program could simply search for those two strings. However, to make things more interesting, you should realize that REM or ' in quotes doesn't count. That is, using REM as part of an input prompt or in a PRINT statement does *not* appear to be a remark to the computer. For example:

```
10 ' Test of Program REM-OVER
20 REM Will Test REMOVAL of REMARKS
30 ' This Remark will be removed.
40 PRINT:PRINT: REM This one will be
   removed.
50 PRINT"This REMARK: REM Will NOT be
   removed."
60 PRINT"This one won't":REM This
   one will.
40 PRINT:PRINT:
50 PRINT"This REMARK: REM Will NOT
   be removed."
60 PRINT"This one won't"
```

Fig. 2-1. Target program for REM-over.

```
10 PRINT"This is NOT a REMark.":REM But this IS.
```

REM-over takes care of this stipulation by simply looking at each program line for quotation marks as well as remarks. If a REM appears after one quote, but before the second, then it is contained within the quotation marks. (This assumes that the programmer has not mismatched quotes, and has included two for every prompt.)

The program, Figs. 2-3 and 2-4, begins by asking the operator for the filename of the program which will have its remarks REM-oved. This filename, F$, is used to form the filename of the output file automatically. First, in line 100 the second filename, F1$, is formed by adding "/REM" onto it. If the filename happens not to have an extension, as, for example, when F$="TEST", then the new filename "TEST/REM" will be legal. A check is made later in line 100 to see if this is so. F1 is equal to INSTR(F$,"/"). If F1=0, that is, F$ does *not* contain a slash and extension, then the program goes to line 110.

However, if a slash *is* found and F1 does *not* equal zero, then the portion of the filename up to the "/" (LEFT$(F$,F1-1)) is taken,

```
40 PRINT:PRINT:
50 PRINT"This REMARK: REM Will NOT
   be removed."
60 PRINT"This one won't"
```

Fig. 2-2. Target program with remarks REM-oved.

A$	Line of program loaded from disk.
B$	Middle string of program line.
F$	Filename of program being processed.
F1$	Filename of output file.
N	Loop counter.
P	Position to begin INSTR search.
Q1	Position of first quote mark.
Q2	Position of second quote mark.
R	Position of remark.
T$	String remaining after remark deleted.

Fig. 2-3. Variables used in REM-over.

and "/REM" tacked on. Next, both files are opened, and a single line is input in line 140. Variable P, which is the position at which the search for REMs begins, is set equal to one. Thus, the initial search for remarks will begin at the first character of A$.

Because both REM and ' can indicate remarks, two searches must be conducted. First, in line 160, the program checks for,' and, if an apostrophe is found, assigns variable R with the position of the suspected remark. Control then branches to line 200. If no apostrophe is located, the program next checks for "REM", in line 180. If no remark is found, then the program line is already remark-free, and the program branches to line 350.

Possible remark lines are examined further at a routine beginning at line 200. There, Q1 is assigned the value equal to the position of a quote mark. If none is found, a remark has indeed been located, and control passes to line 260. If a quote is detected, then REM-over looks at the rest of the program line, beginning at position Q1+1 for a second quote. That value is Q2. If the position of the remark, R, is less than Q1 (the remark appears *before* the first quote, or is more than Q2 (it appears *after* the second quote), then a remark is verified, and the program goes to line 260.

If neither condition is true, then the alleged remark is actually within the quotes, and is disqualified. The program instead makes P equal to the next position after the second quote (Q2+1), and returns control to line 160 to see if any possible remarks exist after position P. In this way, an entire multi-statement line can be looked at, section by section, to detect all remarks.

When a valid remark is located, the program takes all of the program line up to the remark itself, using A$=(LEFT$,R-1), as in line 260. This, in effect, truncates the program at the remark.

We are not finished yet. After all, some program lines consist

```
10 ' ****************
20 ' *                *
30 ' *   REM-over     *
40 ' *                *
50 ' ****************
60 CLEAR 5000

65 ' *** Enter filename ***
70 CLS:PRINT:PRINT
80 PRINTTAB(9)"Enter name of program to have REMARKS removed:"
90 LINEINPUT F$
100 F1$=F$+"/REM":F1=INSTR(F$,"/"):IF F1=0 THEN GOTO 110 ELSE
   F1$=LEFT$(F$,F1-1)+"/REM":GOTO 110
110 OPEN "I",1,F$
120 OPEN "O",2,F1$
130 IF EOF(1) GOTO 380

135 ' *** Load Program Line ***

140 LINEINPUT #1,A$
150 P=1
```

Fig. 2-4. Program listing for REM-over.

```
155 ' *** Check for REMARKS ***

160 R=INSTR(P,A$,"'")
170 IF R<>0 GOTO 200
180 R=INSTR(P,A$,"REM")
190 IF R=0 GOTO 350

195 ' *** Find Quotes, if Any ***

200 Q1=INSTR(P,A$,CHR$(34)):IF Q1=0 GOTO 260
210 Q1=Q1+1
220 Q2=INSTR(Q1,A$,CHR$(34))
230 IF R<Q1 OR R>Q2 GOTO 260
240 P=Q2+1
250 GOTO 160

255 ' *** Strip off REMARKS ***

260 A$=LEFT$(A$,R-1)
270 FOR N=1 TO LEN(A$)
280 B$=MID$(A$,N,1)
290 IF ASC(B$)<48 OR ASC(B$)>57 GOTO 310
```

```
300 NEXT N

310 T$=MID$(A$,N)
320 IF T$="" GOTO 130
330 IF T$=" " GOTO 130
340 IF RIGHT$(A$,1)=":" THEN A$=LEFT$(A$,(LEN(A$)-1))
345 ' *** If line not empty, print to disk ***

350 PRINT A$
360 PRINT#2,A$
370 GOTO 130
380 CLOSE

385 ' *** Again? ***

390 PRINT:PRINT
400 PRINTTAB(21)"Process another file?"
410 PRINTTAB(29)"(Y/N)"
420 A$=INKEY$:IF A$="" GOTO 420
430 IF A$="Y" OR A$="y" THEN RUN ELSE CLS
```


Fig. 2-4. Program listing for REM-over. (Continued from page 11.)

of just a line number and a remark. Cutting off the remark leaves only the line number. This is a bit untidy, and a waste of computer memory as well. So, the program cycles through a FOR-NEXT loop from 1 to the length of A$. Each time through, the string variable B$ is assigned the value of the middle character at position N. This character is checked to see that it is a number in the range 0-9, since all program lines begin with numbers. As soon as B$ does NOT equal a number, REM-over knows that the line number is complete, and control drops down to line 310.

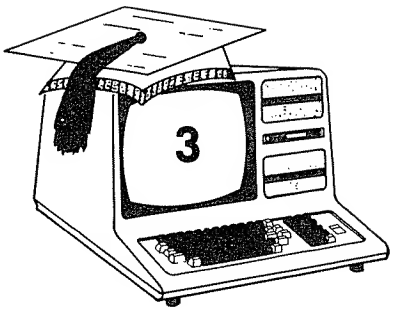
There, T$ is assigned the rest of A$. If T$ is empty, or consists only of a space, then the program knows it has found an "empty" program line, and loops back to line 130 *without* printing anything to the disk. That line has been deleted from the program entirely.

If T$ does have characters, a check is made to see if the final character is a colon, as would be the case if a remark followed a colon on a multi-statement line:

```
10 PRINT"HELLO":REM This is a remark.
```

If a colon is the last character, it is deleted in line 340. Then A$ is printed to the screen, so the operator can monitor the progress of the program, and also printed to the disk. Control goes back to line 130, where a check for the end-of-file is made, and another program line input from the disk.

That's all there is to REM-oving the REM-arks from your programs.



Titler

Now we are ready for some real action. Making a few simple changes in an existing program is kid stuff compared with the "real" thing. That is, generating a new, never-before-existing program line from your very own parameters. That's the function of Titler. This program generates program title blocks like the one shown in Fig. 3-1, which can be merged with your own programs. You don't have to tediously write the program lines yourself, format the title block, or even supply your name and address every time. The program will do that for you. As an added feature, your friends can also use the program by supplying their own names.

This last capability is the result of what are known as "default" values. That is, the programmer defines the contents of variables (Fig. 3-2) storing name, address, and city. Every time the program is run, you simply hit ENTER when asked whether or not a new name and address should be input. (The question is posed in line 120, Fig. 3-3.) Then an INKEY$ loop repeats until the operator presses a key, or hits ENTER. If "N" or ENTER (CHR$(13)) was pressed, then the program drops down to line 280; N$, AD$, and CT$ remain as they were defined in lines 70-90. The default values are used.

If "Y" or some other key is pressed, however, the program will ask for a name, address, city, state, and zip, and then assemble the string variables N$, AD$, or CT$ on its own. In that way a regular

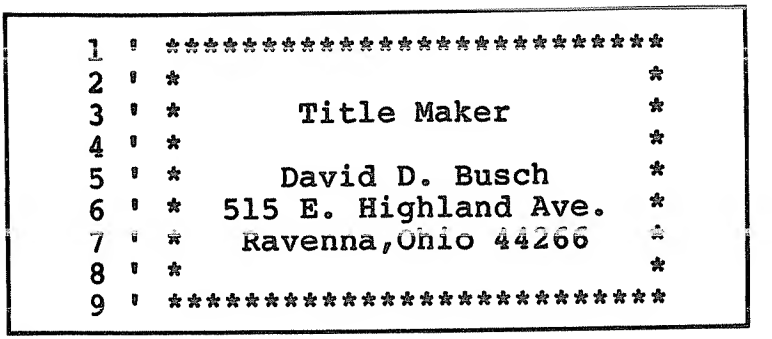


Fig. 3-1. Sample title produced by Titler.

user can be accommodated, while leaving a path open for a friend to use the program as well.

Next, the user is asked for the title of the program, and this is stored in TITLE$. The program checks to see which of the four strings—TITLE$, name, address, or city—is the longest and will thus determine how wide the title block will be. This width, A, is defined in line 350 as the length of the longest string, plus four. The extra four characters will leave room for a space at each end of the longest string, plus an asterisk used as the border.

A disk file named "Title" is opened, and a subroutine at line 760 is accessed to produce a string equal to the next line number to be used in our miniprogram. This routine increments a counter, LC,

A$	Used in INKEY$ loop.
A	Length of widest line in title.
AD$	User's address.
B	Difference between length of line to be incorporated in title and A.
B1	Number of spaces before line.
B2	Number of spaces after line.
C$	User city.
CT$	Name of user's city, state, zip.
LC	Line counter
LN$	Program line currently being built.
N	Loop counter.
S$	User state.
TITLE$	Title of program.
Z$	User zip code.

Fig. 3-2. Variables used in Program Titler.

```
10 ' ********************
20 ' *                  *
30 ' *  Program Titler  *
40 ' *                  *
50 ' ********************
60 CLEAR 1000

65 ' *** Defaults ***

70 N$="Your Name Here"
80 AD$="Your Address Here"
90 CT$="Your City, State, Zip"
100 CLS:PRINT
110 PRINTTAB(23)"Title Block Writer"
120 PRINT:PRINTTAB(20)"Enter Name and Address?"
130 PRINTTAB(15)"(Just Hit <ENTER> to use Defaults)"
140 A$=INKEY$:IF A$=""GOTO 140
150 IF A$=CHR$(13) OR A$="N" OR A$="n" GOTO 280

155 ' *** Enter Name, etc. ***

160 CLS:PRINT
```

Fig. 3-3. Listing for Program Titler.

```
170 PRINTTAB(26)"Enter name :"
180 INPUT N$
190 PRINTTAB(24)"Enter Address :"
200 INPUT AD$
210 PRINTTAB(26)"Enter City :"
220 INPUT C$
230 PRINTTAB(25)"Enter State :"
240 INPUT S$
250 PRINTTAB(24)"Enter Zip Code :"
260 INPUT Z$
270 CT$=C$+","+S$+" "+Z$
280 CLS:PRINT
290 PRINTTAB(20)"Enter title of program :"
300 INPUT TITLE$
310 A=LEN(TITLE$)
320 IF LEN(N$)>A THEN A=LEN(N$)
330 IF LEN(AD$)>A THEN A=LEN(AD$)
340 IF LEN(CT$)>A THEN A=LEN(CT$)
350 A=A+4

355 ' *** Open Disk file ***

360 OPEN "O",1,"TITLE"
370 CLS
```

```
380 GOSUB 760
390 LN$=LN$+STRING$(A+2,"*")
400 PRINT#1,LN$
410 GOSUB 760
420 LN$=LN$+"*"+STRING$(A,32)+"*"
430 PRINT#1,LN$
440 GOSUB 760
450 B=A-LEN(TITLE$):B1=INT(B/2):B2=B-B1
460 LN$=LN$+"*"+STRING$(B1,32)+TITLE$+STRING$(B2,32)+"*"
470 PRINT #1,LN$
480 GOSUB 760
490 LN$=LN$+"*"+STRING$(A,32)+"*"
500 PRINT#1,LN$
510 GOSUB 760
520 B=A-LEN(N$):B1=INT(B/2):B2=B-B1
530 LN$=LN$+"*"+STRING$(B1,32)+N$+STRING$(B2,32)+"*"
540 PRINT#1,LN$
550 GOSUB 760
560 B=A-LEN(AD$):B1=INT(B/2):B2=B-B1
570 LN$=LN$+"*"+STRING$(B1,32)+AD$+STRING$(B2,32)+"*"
580 PRINT#1,LN$
590 GOSUB 760
600 B=A-LEN(CT$):B1=INT(B/2):B2=B-B1
```

Fig. 3-3. Listing for Program Titler. (Continued from page 17.)

```
610 LN$=LN$+"*"+STRING$(B1,32)+CT$+STRING$(B2,32)+"*"
620 PRINT#1,LN$
630 GOSUB 760
640 LN$=LN$+"*"+STRING$(A,32)+"*"
650 PRINT#1,LN$
660 GOSUB 760
670 LN$=LN$+STRING$(A+2,"*")
680 PRINT#1,LN$
690 CLOSE

695 ' *** Final Instructions ***

700 CLS:PRINT
710 PRINT"Renumber your target program so that first"
720 PRINT"line number is higher than 10, then type"
730 PRINTTAB(29)"MERGE ";CHR$(34);"TITLE";CHR$(34);"."
740 PRINT
750 END

755 ' *** Increment Line numbers ***

760 LC=LC+1:LN$=STR$(LC)+"'"+CHR$(32)
770 RETURN
```

Fig. 3-3. Listing for Program Tilter. (Continued from page 19.)

each time it is called. LN$ is then formed by converting the counter LC to a string value, and adding an apostrophe (because our title block will consist of remarks) and a space, CHR$(32). Then the subroutine RETURNs to the main program.

There, LN$ is first added to a string equal to A+2 in length, consisting entirely of asterisks. So, the first line might look something like this:

```
1'**********************
```

That line is PRINTed to the disk in line 400. Then the subroutine at 760 is called again, and a new line is similarly formed. This line consists of a line number one greater than the last, the apostrophe, an asterisk followed by a string of spaces equal to A, and another asterisk. This line will look like this:

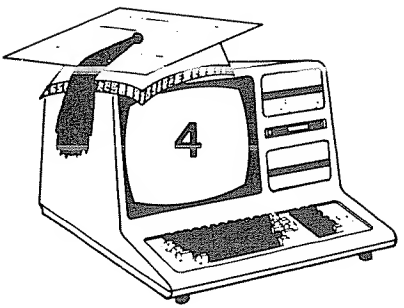
```
2 ' *                    *
```

The following line, containing the title itself, will have an asterisk, some spaces, the title, some more spaces, and another asterisk. The number of spaces will be divided as equally as possible fore and aft, so that the title will be centered. These spaces are calculated by subtracting the length of the title from A, dividing the result by 2, and assigning that value to the number of spaces preceding the title, B1. The number of spaces following is the number remaining after subtracting B1 from B. This is done, instead of simply dividing B by two, because the result will not always be even. It is sometimes necessary to make B1 one space larger than B.

This centering procedure is repeated every line when the name, address, and city are included in the title block. The block is finished when a program line identical to line 1 is written to the disk.

The last step is to close the file. The user is instructed to renumber the target program so that the first line number is larger than 10, and then MERGE it with the TITLE file.

We have created a program from nothing. Next, things get a little more complicated.



Documenter

Although I tried to impress you in the last chapter with the power of Titler, the program was actually small potatoes. A bunch of program lines consisting of nothing more than remarks isn't a real program. After all, a title block doesn't even *do* anything. "Documenter," on the other hand, writes real programs that really do something. Best of all, your own input is kept to a minimum.

One of the difficulties of writing software is developing documentation. At one end of the spectrum is the simple, interactive, self-prompting program with multiple error traps that can be run by the unsophisticated user with no instruction at all. The other extreme is complex software, such as disk operating systems or compilers, that require entire books or manuals to use properly.

Documenter is intended for the broad range between, i.e., those programs that require a few pages of quick instructions at the beginning of the program to make running the software a little simpler. The best part about this program is that it will format and page instructions automatically, write the necessary code, and then append them onto the beginning of your BASIC program. Procedure for using the program is as follows:

1) Renumber your BASIC program so that the starting line number is at least 100, and preferably 200 or higher. This insures that your code will not overlap the instructions tacked onto the beginning.

2) Save the program to disk in ASCII form. Then, load and run

Documenter with a disk containing the target program in one of the disk drives.

3) Input instructions as desired. These may be typed in normally, with backspace used to rub out previous characters. However, a graphics block appears on the screen in the upper right corner. This block indicates the 50-character mark, the "hot zone" that approaches the right hand margin used in the instruction output. If a word ends within this zone, the program will automatically drop down to the next line. If you see that the word you are typing will extend more than three or four characters past the beginning of the hot zone, select an appropriate place for hyphenation, and insert a hyphen. The program will recognize this character and drop down a line at that point.

4) When all copy for instructions has been input, enter an ampersand, which was chosen as a control character.

5) At this point, the program will create a group of directions, an instruction set, so to speak, from your input. These will be in proper program form as listed at the end of this chapter. When complete, the instructions will be appended to the target program you specified when Documenter was run.

6) Line 1 of the new program should be deleted, because it contains the MERGE instructions. The program can then be run as desired. Several pages of your instructions, developed to your own specifications, will be displayed prior to the initial program lines of the original software.

Variables used in Documenter are shown in Fig. 4-1, and the program (listed in Fig. 4-2) works as follows. The user first specifies the name of the previously stored (in ASCII) target program. This is input into a string variable, TP$. Next, a subroutine

A$	Character input from keyboard through INKEY$
B$	String of characters input as copy.
C	Counter for string array PROG$(n)
L$(n)	Border array.
LN$	Stores program lines as compiled.
N	Loop counter.
N1-N6	Loop counters.
P$(n)	Print @locations.
PROG$(n)	Finished program.
TP$	Target program.

Fig. 4-1. Variables used in Documenter.

```
10 ' ***************
20 ' *             *
30 ' * Documenter *
40 ' *             *
50 ' ***************
60 CLEAR 8000
70 DIM L$(12),PROG$(100),LN$(100)
80 C=1

85 ' *** Instructions ***

90 CLS:PRINT
100 PRINT "Your target program must have been saved in
   non-compressed"
110 PRINT "(ASCII) format, using the ',A ' option."
120 LINEINPUT "Enter name of target program  :";TP$

125 ' *** READ BORDER ROUTINE INTO ARRAY ***

130 :  FOR N=1 TO 10
140 :    READ L$(N)
150 :  NEXT N
```

```
155 ' *** READ 'PRINT @ ' LOCATIONS INTO ARRAY ***

160 :  FOR N1=1 TO 6
170 :    READ P$(N1)
180 :  NEXT N1
190 DATA "FOR N=1 TO 63","PRINT @ N,CHR$(159);","NEXT N"
200 DATA "FOR N=833 TO 895","PRINT @ N,CHR$(190);","NEXT
   N","FOR X=0 TO 41","SET(1,X)","SET(127,X)","NEXT X"
210 DATA 136,200,264,328,392,456

215 ' *** INPUT INSTRUCTION COPY ***

220 CLS:PRINT
230 PRINT "Enter instructions now.  Enter ' & '  to quit "
240 PRINT
250 PRINT @ 114,CHR$(191)
260 A$=INKEY$:IF A$="" GOTO 260
270 IF A$=CHR$(8) THEN B$=LEFT$(B$,LEN(B$)-1):GOTO 310
280 IF A$=CHR$(13) THEN GOTO 340
290 IF A$="&" GOTO 390
300 B$=B$+A$
```

Fig. 4-2. Program listing for Documenter.

```
310 PRINT A$;
320 IF LEN(B$)<50 GOTO 260

325 ' *** STORE B$ IN PROG$(N) ARRAY ***

330 IF A$=CHR$(32) OR A$=CHR$(45) GOTO 340 ELSE 260
340 PROG$(C)=B$
350 PRINT CHR$(29);CHR$(26);
360 B$=""
370 C=C+1
380 GOTO 260

385 ' *** BUILD FIRST LINES OF MERGER PROGRAM ***

390 PROG$(C)=B$
400 CLS
410 GOSUB 660
420 LN$(1)="1 MERGE "+CHR$(34)+TP$+CHR$(34)+":STOP:       REM
   DELETE THIS LINE AFTER MERGER"
430 LN$(2)="2 CLS"
440 :  FOR N2=1 TO 11
450 :    LN$(N2+2)=STR$(N2+2)+" "+L$(N2)
```

```
460 :  NEXT N2
470 LN=N2

475 ' *** BUILD 'PRINT @ ' LINES WITH EACH ELEMENT ***

480 :  FOR N3=1 TO C STEP 6
490 :    FOR N4=0 TO 5
500 :       GOSUB 660:LN$(LN)=LN$(LN)+"PRINT @ "+P$(N4+1)
          +","          +CHR$(34)+PROG$(N3+N4)+CHR$(34)+";"
510 :    NEXT N4
520 :    GOSUB 660:LN$(LN)=LN$(LN)+"PRINT @ 714,"+CHR$(34)
        +"HIT ANY KEY TO CONTINUE"+CHR$(34)+";"
530 :    GOSUB 660:LN$(LN)=LN$(LN)+"IF INKEY$="+CHR$(34)
        +CHR$(34)+" GOTO "+STR$(LN)
540 :       FOR N5=1 TO 6
550 :         GOSUB 660
560 :         LN$(LN)=LN$(LN)+"PRINT @ "+P$(N5)+","
           +"STRING$(55,32);"
570 :       NEXT N5
580 :  NEXT N3
```


Fig. 4-2. Program listing for Documenter. (Continued from page 25.)

```
590 GOSUB 660:LN$(LN)=LN$(LN)+"IF INKEY$="+CHR$(34)
   +CHR$(34)+"GOTO "+STR$(LN)

595 ' *** SAVE MERGER PROGRAM TO DISK ***

600 OPEN "O",1,"MERGER"
610 :  FOR N6=1 TO LN
620 :    PRINT #1,LN$(N6)
630 :  NEXT N6
640 CLOSE 1

645 ' *** RUN MERGER PROGRAM TO COMBINE TARGET/INSTRUCTIONS ***

650 RUN "MERGER"

655 ' *** INCREMENT LINE NUMBERS AND INITIATE NEW LINE ***

660 LN=LN+1:LN$(LN)=STR$(LN)+" ":RETURN
```

Fig. 4-2. Program listing for Documenter. (Continued from page 27.)

that was parsed and entered as DATA lines is read into a string array, L$(n). This 10-line subroutine prints a graphic border around the screen of the TRS-80. The various portions of the routine, minus line numbers, are stored in data lines at 190-200.

Next, a group of numbers (data line 210) is read into a second array, P$(n). These numbers correspond to six screen locations that will be used in PRINT @ routines to be created later. For example, the first item of data, 136, will be used to PRINT @ 136 either instructional material or blanks to erase that portion of the screen—without disturbing the graphics border around the edges.

At this point, the person running the program is urged to begin entering the instructional copy. The right margin hot zone is first placed on the screen, in line 250. Eventually this will scroll out of sight, but the other lines of copy will constitute a sufficient reminder for most.

An INKEY$ strobing loop in line 260 allows keyboard input of any character. When A$ does not equal null (" "), control drops to line 270, where a check is made to see if the backspace character (CHR$(8)) has been entered. If so, the string that stores the current input, B$, is rubbed out from the right side by one character.

If ENTER has been hit (CHR$(13)), control passes to line 340; otherwise, unless the escape ampersand has been entered, A$ is both added to B$, and printed to the screen. If B$ is not greater than 50 characters, then the program returns to the INKEY$ line for additional input.

When 50 characters have been entered, the time is ripe to find the end of a word. Control always drops to line 330, where Documenter looks for a space, CHR$(32), or a hyphen, CHR$(45). If one is not found, INKEY$ is accessed for more input.

When the length of B$ is more than 50 and a space or hyphen is recognized, then B$ is stored in a string array, PROG$(n). Then B$ is nulled and C (the PROG$(n) counter) is incremented by one; the next time that portion of the program is called, B$ will be deposited one element farther down in the array. This portion of the program is used whenever ENTER is hit during the INKEY$ loop.

Input may continue until "&" is entered. At this point, the last value of B$ is stored in PROG$(C), and construction of the program subroutine begins. Program lines are stored in a string array, LN$(n). These lines are assembled similarly to the titles in the last chapter. That is, first a program line number is produced, and then the necessary parts of the program line are added to that. The first program line is always line number 1, followed by MERGE, a quote

```
1 MERGE "INSTRUCT":STOP:      REM DELETE THIS LINE AFTER MERGER
2 CLS
3 FOR N=1 TO 63
4 PRINT @ N,CHR$(159);
5 NEXT N
6 FOR N=833 TO 895
7 PRINT @ N,CHR$(190);
8 NEXT N
9 FOR X=0 TO 41
10 SET(1,X)
11 SET(127,X)
12 NEXT X
13 PRINT @ 136,"     This program will allow you to enter
instruc-";
14 PRINT @ 200,"tions for a Basic program of your choice.  It
will";
15 PRINT @ 264,"automatically format the directions into lines
of ";
16 PRINT @ 328,"approximately 50 characters each, and append
them ";
17 PRINT @ 392,"with the target program.";
18 PRINT @ 456,"     Simply type; as you enter the 'hot zone'
the ";
```

```
19 PRINT @ 714,"HIT ANY KEY TO CONTINUE";
20 IF INKEY$="" GOTO  20
21 PRINT @ 136,STRING$(55,32);
22 PRINT @ 200,STRING$(55,32);
23 PRINT @ 264,STRING$(55,32);
24 PRINT @ 328,STRING$(55,32);
25 PRINT @ 392,STRING$(55,32);
26 PRINT @ 456,STRING$(55,32);
27 PRINT @ 136,"program will drop down a line when you reach the
end ";
28 PRINT @ 200,"of a word.  If a word will extend more than a
few ";
29 PRINT @ 264,"characters past the graphics block in the upper
right, ";
30 PRINT @ 328,"you need to insert a hyphen.  The program will
recognize ";
31 PRINT @ 392,"this as well as a space.";
32 PRINT @ 456,"     Your target program must be saved in ASCII
form ";
33 PRINT @ 714,"HIT ANY KEY TO CONTINUE";
34 IF INKEY$="" GOTO  34
```

Fig. 4-3. Sample program produced by Documenter.

```
35 PRINT @ 136,STRING$(55,32);
36 PRINT @ 200,STRING$(55,32);
37 PRINT @ 264,STRING$(55,32);
38 PRINT @ 328,STRING$(55,32);
39 PRINT @ 392,STRING$(55,32);
40 PRINT @ 456,STRING$(55,32);
41 PRINT @ 136,"and have line numbers higher than 200";
42 PRINT @ 200,"";
43 PRINT @ 264,"";
44 PRINT @ 328,"";
45 PRINT @ 392,"";
46 PRINT @ 456,"";
47 PRINT @ 714,"HIT ANY KEY TO CONTINUE";
48 IF INKEY$="" GOTO  48
49 PRINT @ 136,STRING$(55,32);
50 PRINT @ 200,STRING$(55,32);
51 PRINT @ 264,STRING$(55,32);
52 PRINT @ 328,STRING$(55,32);
53 PRINT @ 392,STRING$(55,32);
54 PRINT @ 456,STRING$(55,32);
55 IF INKEY$=""GOTO  55
```

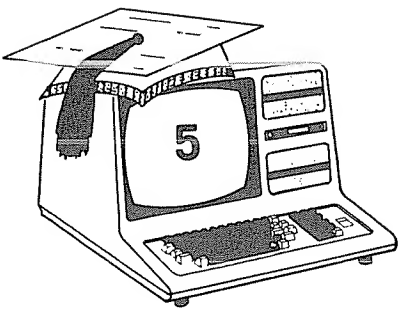
Fig. 4-3. Sample program produced by Documenter. (Continued from page 31.)

mark, and the title of the target program. The second program line is always "2 CLS".

The next few lines are always the lines which write the border routine. They are assembled by adding elements of L$(n) to the program line number produced by a FOR-NEXT loop beginning at line 430.

Nested FOR-NEXT loops next construct the information pages, divided into six lines per page. So, the outer loop steps from 1 to C (total number of lines) STEP 6. The inner loop determines which PRINT @ location is used in that given program line. Each screen has a "Hit any key to continue" line added, and an INKEY$ loop.

If all this is a bit unclear, carefully examine the program listing, which is liberally sprinkled with comments. Also, look at the sample program, Fig. 4-3,, which was produced using Documenter. The steps needed to assemble a finished program using operator input will be explained further in later programs. This is just the beginning.



Tabber

Time for a breather. Tabber is a simple yet elegant little program that will be very useful to you. It creates no new program lines, doesn't make your computer operate 50 percent faster, and won't even make your laundry whiter.

What it will do is automatically center various prompts that are printed to the screen using PRINT or INPUT statements. Instead of sloppy screen formatting, you can have neat copy. It will work with both 64-column screens in the TRS-80 Models I/III, and 80-column screens of the Model 4. Best of all, you need to make only one small change in your programming habits.

To center any prompt, simply type PRINTTAB(T) instead of calculating the proper tab position yourself. See Fig. 5-1. With messages that are going to be PRINTed to the screen, just insert TAB(T). If a program presently includes the prompt after an INPUT or LINEINPUT statement, you will have to do some rewriting, since there is no such thing (yet) as INPUTTAB(n) or LINEINPUTTAB(n) statements for TRS-80 computers. Use the second line, rather than the first in the examples below:

```
WRONG:  10 INPUT "Enter your name:";A$
RIGHT:  10 PRINTTAB(T)"Enter your name:";:INPUT
           A$
```

(Model 4 users should separate PRINT and TAB, as well as LINE and INPUT, with a space.)

```
10 PRINTTAB(T)"THIS PROGRAM DEMONSTRATES THE USE"
20 PRINTTAB(T)"OF TABBER/BAS.  ANY PROGRAM USING"
30 PRINTTAB(T)"THE SPECIAL 'T' TAB WILL HAVE THAT"
40 PRINTTAB(T)"PROMPT CENTERED ON THE SCREEN"
```

Fig. 5-1. Target program to demonstrate Tabber.

You can even run programs using TAB(T) without running them through Tabber. This is especially useful during program development and testing. Simply insert the TAB(T)s as you go along. Until the finished program has been processed by Tabber, all prompts with TAB(T) will be printed flush left, so long as the variable T is not used elsewhere within your program. If not, it will have a default value of zero, and the program will tab zero spaces for each prompt. Then, when the program is done, save it in ASCII form and run Tabber. Tabber will search through each program line. When it finds TAB(T) it will measure the length of the prompt remaining, calculate how many spaces must be tabbed to center that message on a 64- or 80-column screen, and then replace the "T" with an appropriate number. Figure 5-2 shows our target program after Tabber has finished its work.

A few programming tips are included in this program. Menu input routines are one area ripe for improvement. Many programs will offer the operator a choice of actions, listed in a "menu" on the screen. Items from menus can be selected by having the user press the first letter of the menu item name, enter the whole choice, or enter a number that precedes the menu choice.

Having the user type in the whole name is rarely done, because a simple typing error could invalidate an otherwise correct entry. If a person wants a 64-column screen but types 63 instead, it is a shame to make him or her redo the whole entry just for missing by one, or, worse, having the program crash because it doesn't recognize the choice. Entering one character is popular, especially when a menu is accessed frequently. The user can easily memorize which letter triggers which menu choice, because of the mnemonic connection. The following is a typical letter-oriented menu:

(L)oad
(S)ave
(E)xit
(C)ontinue

A problem could occur if two menu choices started with the same letter, and the programmer could not think of a convenient synonym that used another initial letter. In addition, such menus force the

```
10 PRINTTAB(15)"THIS PROGRAM DEMONSTRATES THE USE"
20 PRINTTAB(15)"OF TABBER/BAS.  ANY PROGRAM USING"
30 PRINTTAB(15)"THE SPECIAL 'T' TAB WILL HAVE THAT"
40 PRINTTAB(17)"PROMPT CENTERED ON THE SCREEN"
```

Fig. 5-2. Target program with TABs inserted.

non-typist user to hunt around the keyboard for letters that may be widely separated.

Numeric menus, on the other hand, have choices arranged in neat rows across the top of the keyboard. The limitation is that only 10 menu choices can be listed, if we want single-key entry (0-9). Even then we open ourselves to problems, because the simplest input methods could confuse a null entry (just pressing ENTER, for example) with zero. It is possible to check the CHR$ values of the entries, to differentiate between zero (CHR$(48)) and ENTER (CHR$(13)). One could also extend a numeric menu by using hexadecimal notation, following nine with A, B, C, D, or E.

In practice this is seldom needed. Tabber's menu has only two choices, that between 64- and 80-column formatting. However, it also uses a built-in error trap, something that is too often forgotten by beginning programmers. Some will write a menu routine like this:

```
10  PRINT"1.) Load program"
20  PRINT"2.) Save program"
30  INPUT"Enter Choice";CH
40  ON CH GOTO 100,200
```

Now, if a naive user enters "L" or "S", or some other letter by mistake, a cryptic "REDO FROM START" message will be displayed. That is of no help at all. Entering a number larger than two will send the program to the line following 40, whatever *that* is. This could crash the whole program. We can avoid the REDO message by using CH$ instead of CH in the INPUT, since strings will accept letters as well as numbers. Converting to numerics, e.g., CH=VAL(CH$), will send us to our ON CH GOTO line happily—except we still haven't handled the inappropriate input that might result. It is also necessary for the user to remember to hit ENTER before the input is accepted. The user either has to be sophisticated enough to do this on his or her own, or else we have to waste a line to prompt the user to do so.

Since all we want is a single character, why not use INKEY$ to

get it? Then, if the character is not valid, just send control back to the INKEY$ loop until a proper entry *is* made. That is what is used in "Tabber," the variables and listings for which are shown in Figs. 4-3 and 4-4. Line 120, for example, is an INKEY$ loop that repeats until a character is pressed. That character, A$, is converted to a number value, A, in line 130. If A<1 or A>2, the program loops back; otherwise, it sets the value of S to either 64 or 80, as appropriate.

Next, the user enters the filenames for the input and output files, and a single line is loaded from disk, in line 230. The next line looks for an occurrence of "TAB(T)" in the target program line. Since the string "TAB(T)" is likely to be unique, no effort is made to check if it is contained in quotes, or after a remark. Odds are that it will never appear in your program, except where you actually do want to center a prompt. This is mentioned because Tabber did "crash" when it was used to process itself—caused by line 240, in which TAB(T) is contained as part of the program itself, and not before any prompt. In all other cases, TAB(T) will be followed by a prompt and a matched pair of quote marks, but in this case that was not true.

Whenever Tabber finds TAB(T), it looks for the position of the first quote, loads the value of the rest of the program line after that quote, and then cuts off the line following the second quote (line 280). B$ will then contain only the material in the prompt.

The next step is to measure the length of the prompt, subtract that from S, which is the screen width (either 64 or 80 columns), and divide by 2. The resulting number, D, is the number of spaces that should be tabbed to center the prompt.

A new program line is then assembled in line 310, taking everything that appears *before* the TAB(T), adding that to a string

A$	Program line being examined.
B$	Portion of program line.
C	Position of "TAB(T)" in program line.
C1	Position of quote in program line.
D	Half the difference between prompt length and display line length.
D$	Amount to tab, added to program line.
F$	File to be processed.
F2$	Output file.
S	Length of display line, either 64 or 80.

Fig. 5-3. Variables used in Tabber.

```
10 ' **********************
20 ' *                    *
30 ' *    Tabber          *
40 ' *                    *
50 ' **********************
60 CLEAR 1000
70 CLS:PRINT:PRINT
80 PRINT TAB(13)"IS PROGRAM FOR 64 OR 80 COLUMN SCREEN?"
90 PRINTTAB(20)"1.) 64 COLUMN"
100 PRINTTAB(20)"2.) 80 COLUMN"
110 PRINT:PRINT TAB(20)"ENTER CHOICE :"
120 A$=INKEY$:IF A$="" GOTO 120
130 A=VAL(A$)
140 IF A<1 OR A>2 GOTO 120
150 IF A=1 THEN S=64 ELSE S=80

155 ' *** Enter Name of File to Process ***

160 CLS:PRINT:PRINT
170 PRINT TAB(12)"ENTER PROGRAM WITH TABS TO BE CENTERED:"
180 LINEINPUT F$
190 PRINT TAB(18)"ENTER NAME OF OUTPUT FILE :"
```

```
200 LINEINPUT F2$
210 OPEN "I",1,F$
220 OPEN "O",2,F2$

225 ' *** Load a Line ***

230 LINEINPUT#1,A$
240 C=INSTR(A$,"TAB(T)")
250 IF C=0 GOTO 330
260 C1=INSTR(C,A$,CHR$(34))+1
270 B$=MID$(A$,C1)
280 B$=LEFT$(B$,INSTR(B$,CHR$(34))-1)
290 D=INT((S-LEN(B$))/2)
300 D$=MID$(STR$(D),2)
310 A$=LEFT$(A$,C+3)+D$+MID$(A$,C+5)
320 GOTO 240

325 ' *** Print to Disk ***

330 PRINT#2,A$
```

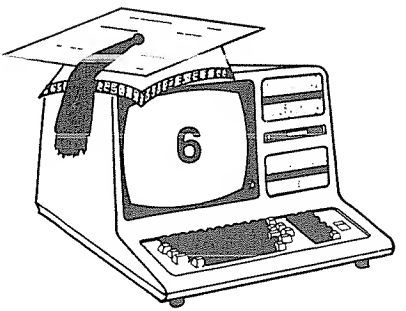
Fig. 5-4. Program listing for Tabber.

```
340 PRINT A$
350 IF EOF(1) GOTO 370
360 GOTO 230
370 CLOSE
380 CLS:PRINT:PRINT
390 PRINT TAB(27)"FINISHED."

395 ' *** Again? ***

400 PRINT:PRINT
410 PRINTTAB(21)"Process another file?"
420 PRINTTAB(29)"(Y/N)"
430 A$=INKEY$:IF A$="" GOTO 430
440 IF A$="Y" OR A$="y" THEN RUN ELSE CLS
```

Fig. 5-4. Program listing for Tabber. (Continued from page 39.)

representation of the tab value (the leading space has been deleted in line 300), and finishing off with the rest of the program line, beginning with ")". Thus, the "T" has been deleted and replaced with a number. The program then loops back to line 240 to see if any more TAB(T)'s appear in the program line. This allows Tabber to process multiple TAB(T)s appearing on a single line.

Once the work is finished, or if a line contains no TAB(T)s in the first place, control drops down to line 330, where A$ is printed to disk and screen. A check is made in line 350 to see if the end-of-file has been reached. If not, the program loops back to line 230 to load another program line from disk. Otherwise, the processing is finished.



Screen Editor

The next three programs in the book, Screen Editor, DB Starter, and Proofer, make up a trilogy (of sorts) called "Automatic Programmer." The three in the Automatic Programmer series are related programs that *might* be thought of as integrated, but they aren't. No data files are transferable from one to the other. However, output from one of three can be processed or combined with output from the others quite easily.

These programs are an attempt to present some professional programming concepts, showing how error traps, help screens, instructional files, etc., can enable programs to be self-documenting and usable even by the neophyte.

All three make use of a fourth program, "Automatic Programmer Documentation," which serves as a help file and introduction to all three. It also is a menu of sorts that can be used to load and run one of the other programs.

The first of the Automatic Programmer series is Screen Editor, which you will find to be one of the most useful programs in this book. I relied heavily on it to write instructional screens for many of the other programs here, and even for itself. With a few minor changes, the program is compatible with the Microsoft BASIC Compiler. A much faster-running compiled version was used, cutting programming time down from a minute or two to a few seconds.

Have you ever wished that you could design your program

menus, instruction screens, and other CRT displays with a word processor or some similar program—and then tell your TRS-80 something like, "Hey, I want my screen output to look like this. Please write a few lines of code for me that will reproduce this in my program."

"Screen Editor" will do exactly that for you. Use it as a screen-oriented text editor to lay out your display exactly as you want it to appear. Then specify a beginning line number, line number increment, and a filename for the finished code. The program will then write a suitable subroutine that can be MERGEd with an existing program to produce the desired display.

Ordinary, line-oriented program input and editing are somewhat tedious when neat, nicely formatted screen layout is desired. It is necessary to use a copy of the TRS-80 Model I/III or 4 screen map, and do a great deal of laborious notation on PRINT@locations, or POKEs to video memory. Even less complicated layouts require calculating TAB positions and other time-consuming tasks. Consider the work that would be involved in programming a display to provide the menu in Fig. 6-1.

With Screen Editor, simply use the arrow keys to move the cursor around on the full screen. Press character keys to place alphanumerics where desired. The layout can be quickly done by eye. Then, hit ENTER, specify what line numbers are desired for this subroutine, and collect a finished program module like Fig. 6-2 from your disk a few minutes later. There, stored in ASCII form and ready for merging will be 16 program lines that reproduce what you designed on the screen. Instead of 15 or 20 minutes of coding, RUNning the program to check the appearance of the output, making changes, and so forth, you have three to five minutes of typing with a wordprocessor-like tool.

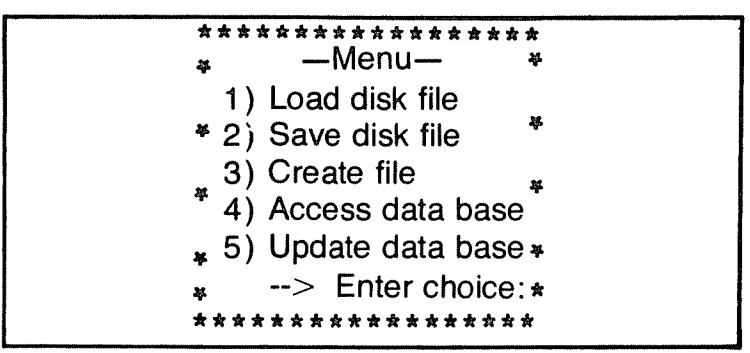


Fig. 6-1. A typical program menu.

```
 20  CLS
 30 PRINT TAB( 5)"*************** "
 40 PRINT TAB( 5)"*              * "
 50 PRINT TAB( 5)"*   SCREEN     * "
 60 PRINT TAB( 5)"*   EDITOR     * "
 70 PRINT TAB( 5)"*              * "
 80 PRINT TAB( 5)"*************** "
 90 PRINT
100 PRINT TAB( 25)"############################## "
110 PRINT TAB( 25)"#  TYPICAL EXAMPLE SCREEN # "
120 PRINT TAB( 25)"############################## "
130 PRINT
140 PRINT
150 PRINT TAB( 14)"This is a screen prepared by
the Screen Editor "
160 PRINT
170 PRINT
180 PRINT
"";
200   A$=INKEY$:IF A$="" GOTO 200
```

Fig. 6-2. Example of program produced by Screen Editor.

The trick is accomplished by PEEKing into video memory, noting what character (if any) has been placed there by the user, and then assigning each screen line to a separate element of a string array, L$(n). Then, each of the elements in L$(n) are used to assemble an appropriate program line which PRINT the entire line to the user's screen. If, say, line one consists of four spaces, fifty-six asterisks, and four more spaces, that entire line will be PRINTed in the resulting program. No PRINT@'s or other calculations need to be made.

Screen Editor, in other words, reproduces your screen arrangement, spaces and all. It may not be the most memory-efficient way of invoking a desired screen within your program, but for disk users with 32K or 48K of memory available, the waste will be negligible compared to the time saved.

Actually, a nifty technique is used to eliminate the leading and trailing spaces. As the program looks at each video line in turn, it sets a BFLAG when it encounters the first non-space character, and an EFLAG when it encounters the last non-space character on the line.

In assembling the finished program lines, it constructs a PRINTTAB statement that tabs to the position of the first non-space. The following characters, spaces and all, are reproduced until the last non-space, when a closing quote is added. Thus, a line like:

Hello!

Would not be turned into a program line like this:

```
10 Print"        Hello!                                    "
```

Instead, the line would read:

```
10 PRINTTAB(10)"Hello!"
```

The program is divided into two main sections. The first allows user input of the screen design. An INKEY$ keyboard strobing loop looks for input (line 120). If ENTER has been pressed (CHR$(13)), control drops down to the video memory peeking/program assembly section. Otherwise, Screen Editor looks at the character input to see if it was an arrow key (character strings 8, 9, 10, and 91). If so, one of four subroutines which move the cursor in the indicated direction are accessed.

The cursor is not allowed to move off the top of the screen, nor past the 15th line of the display. To check for this condition, each subroutine first looks to see whether or not the proposed position

A$	Character input from keyboard, through INKEY$
B	Beginning of video memory.
C	Cursor character.
CU	Counter
E	End of video memory.
EFLAG	End of character line flag.
F$	Filename of output file.
IC	Increment to increase line number by.
L$	Endof line character, either ";" or ""
LN$(n)	Stores finished program lines.
LN$	Program line currently being built.
N	Loop counter.
N1-N9	Loop counters.
PR$	Program line being read from video memory.
S9	Position in filename of "/"
SP	Space (CHR$(32)) as a cursor character.
t	Value PEEKed in video memory.
Z	Position of cursor.
Z1	Check to see if middle of screen reached.

Fig. 6-3. Variables used in Screen Editor.

```
10 '      ****************************
20 '      *                          *
30 '      *       Screen Editor      *
40 '      *                          *
50 '      ****************************
60 '
70 CLEAR 10000
80 DEFINT A-Y
90 ON ERROR GOTO 1570
100 DIM LN$(400)
110 GOTO 140
120 A$=INKEY$:IF A$="" GOTO 120
130 RETURN
140 B=15360:E=16319:CU=1:Z=B:C=43:SP=32
150 LA$=STRING$(64,"*")

155 ' *** Instructions? ***

160 CLS
170 GOSUB 330
180 GOSUB 200
190 GOTO 240
200 :  FOR N8=1 TO 3
```

```
210 :    PRINT"*";TAB(63)"*";
220 :  NEXT N8
230 RETURN
240 PRINT"*";TAB(17);"-- Do you want instructions ? --";
    TAB(63);"*";
250 GOSUB 200
260 PRINT"*   You may also type 'H' or 'HELP' to most input
   prompts.     *";
270 PRINT STRING$(64,"*");
280 GOSUB 120
290 IF A$="N" OR A$="n" THEN CLS: GOTO 460
300 IF A$="H" OR A$="h" THEN RUN"AUTOPROG/DOC"
310 IF A$="Y" OR A$="y" THEN RUN"AUTOPROG/DOC" ELSE 280
320 CLS
330 PRINT"**************************************************
    ***************";
340 PRINT"*
    *";
350 PRINT"*                      Automatic Programmer
             *";
360 PRINT"*                       Screen Editor
             *";
```

Fig. 6-4. Program listing for Screen Editor.

```
370 PRINT"*                          By:     David D. Busch
            *";
380 PRINT"*      ------------------------------------------------
    -----------  *";
390 RETURN
400 CLS
410 CLOSE
420 CU=1
430 :  FOR N8=1 TO 100
440 :    LN$(N8)=""
450 :  NEXT N8
460 LN=10:IC=10
470 PRINT:PRINT:PRINT
480 GOSUB 500
490 GOTO 610

495 ' *** Enter filename of screen ***

500 LINE INPUT"ENTER FILE NAME   :  ";F$
510 IF LEFT$(F$, 4)="HELP" OR F$="H" OR F$="h" GOSUB 1690
520 S9=INSTR(F$,"/")
530 IF S9=0 GOTO 580
540 IF LEN(MID$(F$, S9))>4 GOTO 500
```

```
550 IF VAL(MID$(F$, S9+1, 1))>0 GOTO 500
560 IF LEN(F$)>12 GOTO 500
570 GOTO 610
580 IF LEN(F$)>8 GOTO 500
590 IF F$="" GOTO 500
600 RETURN
610 IF F$="" THEN F$="TEST"

615 ' *** Instructions ***

620 CLS:PRINT
630 PRINT TAB(13)"CUSTOM SCREEN DESIGNING MODULE "
640 PRINT
650 Z=B
660 PRINT TAB(5)"Use the arrow keys to move the '+' cursor
   around  the "
670 PRINT"screen.  You may enter any alphanumeric characters or
   symbols. "
680 PRINT"Only the first fifteen lines of the screen may be
   used.   A "
690 PRINT"large graphics block replaces the cursor when at the
   exact "
```


Fig. 6-4. Program listing for Screen Editor. (Continued from page 47.)

```
700 PRINT"center of the screen, as an aid to centering. (Does
   not appear "
710 PRINT"when <space> bar is used instead of right arrow key
   to position)";
720 PRINT"You may write over any text on the screen, draw
   borders, etc. "
730 PRINT"When you are satisfied, hit <ENTER>.   Screen will be
   painted "
740 PRINT"white while your data are captured. "
750 PRINT:PRINT
760 PRINT TAB(15)"-- HIT ANY KEY TO BEGIN -- "
770 GOSUB 120
775 ' *** Look for keyboard input ***
780 CLS
790 GOSUB 120
800 IF A$=CHR$(13)POKE Z, SP: GOTO 1210
810 IF A$=CHR$(91)THEN 890
820 IF A$=CHR$(10)THEN 970
830 IF A$=CHR$(9)THEN 1050
840 IF A$=CHR$(8)THEN 1130
850 A=ASC(A$)
855 ' *** Move Cursor ***
```

```
860 POKE Z, A
870 IF Z+1<E THEN Z=Z+1: POKE Z, C
880 GOTO 790
890 IF Z-64<B THEN 790
900 POKE Z, SP
910 Z=Z-64
920 Z1=Z-15360
930 IF Z1/32=INT(Z1/32)THEN C=191
940 POKE Z, C
950 C=43
960 GOTO 790
970 IF Z+64>E THEN 790
980 POKE Z, SP
990 Z=Z+64
1000 Z1=Z-15360
1010 IF Z1/32=INT(Z1/32)THEN C=191
1020 POKE Z, C
1030 C=43
1040 GOTO 790
1050 IF Z+1>E THEN 790
1060 POKE Z, SP
1070 Z=Z+1
```

Fig. 6-4. Program listing for Screen Editor. (Continued from page 49.)

```
1080 Z1=Z-15360
1090 IF Z1/32=INT(Z1/32)THEN C=191
1100 POKE Z, C
1110 C=43
1120 GOTO 790
1130 IF Z-1<1 THEN 790
1140 POKE Z, SP
1150 Z=Z-1
1160 Z1=Z-15360
1170 IF Z1/32=INT(Z1/32)THEN C=191
1180 POKE Z, C
1190 C=43
1200 GOTO 790
1205 ' *** Peek Screen Routine ***
1210 GOSUB 1520
1220 LN$(CU)=LN$(CU)+"CLS"
1230 LN=LN+IC
1240 CU=CU+1
1250 :  FOR N=0 TO 1023 STEP 64
1260 :    BFLAG=0
1270 :    EFLAG=0
1280 :    N3=0
```

```
1290 :     PR$=""
1300 :       FOR N1=N TO N+63
1310 :         N3=N3+1
1320 :         T=PEEK(N1+15360)
1330 :         POKE N1+15360, 191
1340 :         IF BFLAG>0 THEN 1360
1350 :         IF T<>32 THEN BFLAG=N3: EFLAG=N3 ELSE 1380
1360 :         PR$=PR$+CHR$(T)
1370 :         IF T<>32 THEN EFLAG=N3
1380 :       NEXT N1
1390 :    IF RIGHT$(PR$, 1)=CHR$(32)THEN PR$=LEFT$(PR$,
         LEN(PR$)-1)
1400 :     IF EFLAG=64 THEN L$=";" ELSE L$=""
1410 :     IF BFLAG=0 THEN 1440
1420 :     LN$(CU)=STR$(LN)+" PRINT TAB(" +STR$(BFLAG-1)+")"
          +CHR$(34)+MID$(PR$, 1, EFLAG-(BFLAG-2))+CHR$(34)+L$
1430 :     GOTO 1450
1440 :     LN$(CU)=STR$(LN)+" PRINT"
1450 :     CU=CU+1
1460 :     LN=LN+IC
1470 :   NEXT N
1480 LN$(CU)=LN$(CU)+CHR$(34)+CHR$(34)+";"
```

Fig. 6-4. Program listing for Screen Editor. (Continued from page 51.)

```
1490 GOSUB 1520
1500 LN$(CU)=LN$(CU)+" "+"A$=INKEY$:IF A$="+CHR$(34)+CHR$(34)+"
    GOTO"+STR$(LN)
1510 GOTO 1770
1520 LN=LN+IC
1530 CU=CU+1
1540 LN$(CU)=STR$(LN)+"  "
1550 RETURN
1555 ' *** Error Trap ***
1560 CLS:PRINT
1570 PRINT:PRINT
1580 PRINT TAB(20)"*****  UNKNOWN ERROR  *****"
1590 PRINT TAB(25)"IN LINE ";ERL
1600 FOR N9=1 TO 500
1610 NEXT N9
1620 RESUME 400
1630 CLS:PRINT:PRINT
1640 RETURN
1650 PRINT
1660 PRINT TAB(15)"Hit any key to resume program"
1670 GOSUB 120
1680 RETURN
```

```
1685 ' *** Help ***

1690 GOSUB 1630
1700 PRINT"You should enter the filename you want  -- it must"
1710 PRINT"be a legal Disk basic name, or your input will be"
1720 PRINT"rejected."
1730 PRINT
1740 LINE INPUT"ENTER FILENAME :";F$
1750 RETURN

1755 ' *** Save Screen to Disk ***

1760 GOSUB 1630
1770 OPEN "O",1,F$
1780 :  FOR N=1 TO CU
1790 :    PRINT#1,LN$(N)
1800 :  NEXT N
1810 CLOSE 1
1820 PRINTTAB(21)"Produce another screen?"
1830 PRINTTAB(29)"(Y/N)"
1840 A$=INKEY$:IF A$="" GOTO 1840
1850 IF A$="Y" OR A$="y" THEN RUN ELSE CLS
```

Fig. 6-4. Program listing for Screen Editior. (Continued from page 53.)

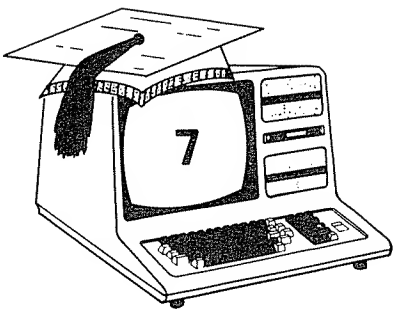
(P) for the cursor would be less than 15360 in video memory (defined as B in line 140), or more than 16314 (defined as E in the same line).

If the new move is okay, then a space (SP=32) is POKED into the old location, and C (CHR$(43), a plus sign) is POKED into the new. Control passes back to the INKEY$ line for further input. If the character entered is not an arrow key, then that character is POKED to the screen.

The second section of the program, beginning at line 1210, PEEKs at the entire video memory, noting what character appears there. Here the program lines, deposited into string array LN$(n), are built. After all the screen has been read, an INKEY$ loop is constructed as a final line, to keep the new program's display on the screen until the user presses a key.

Finally, beginning at line 1760, the elements of LN$(n) are printed to the disk under the filename specified. The finished screen image is captured in program form for you to use in programs of your own.

Like all the programs in the Automatic Programmer series, Screen Editor has many error traps built in. Entering "Help" or "H" to the input prompts will call up the help file, or display a tip. More complex error traps will be discussed in later chapters.



DataBase Starter

For the microcomputer user, the self-programming computer is still some time in the distant future. Or is it? There are three things that computers have a knack for: processing data, controlling functions, and constructing designs from smaller building blocks. The first two are simple enough. Ask a computer to add 367 to 598, and it will happily comply. Tell it to send a signal to port X whenever it receives input from port Y, and a computer will gladly control your carburetor, monitor your house, or keep your Boeing 747 on course. When a human is available to provide a list of criteria and parameters, a computer is entirely capable of combining components from an existing library to assemble, or "design" a complex product.

A computer program is nothing more than a design for accomplishing a desired task. Once a human being has determined how to get from point A to point B, it is entirely practical to have a computer choose from a library of subroutines to put together a program. The next program in the "Automatic Programmer" series is "DB Starter," which illustrates the basic concept.

This program will ask the user for certain program parameters, such as whether a "menu" is needed, whether or not data will be stored in a string array, size of the array, and other information, and then "write" a BASIC program skeleton that conforms to these parameters.

Figure 7-1 is a sample program that was written by DB Starter.

```
20  CLEAR MEM/3*2
30  DATA Name,Address,Phone,Zip
40  DIM DA$( 20, 30),DTA$( 4)
50  NC= 30
60  FOR G=1 TO  4:READ DTA$(G):NEXT G
70  CLS:PRINT:PRINT"   **********  MENU **********":PRINT
80  PRINT" 1.)   Access Data"
90  PRINT" 2.)   Update Data"
100  PRINT" 3.)   Start Database"
110  PRINT" 4.)  LOAD FILE FROM DISK"
120  PRINT" 5.)  SAVE FILE TO DISK"
130  PRINT
140  INPUT"ENTER CHOICE : ";CH$
150  CH=VAL(CH$): IF CH<1 OR CH>  5 GOTO 140
160  ON CH GOSUB 500, 1000, 1500, 2000, 2500
170  GOTO  70
500   REM ****** INSERT Access Data SUBROUTINE HERE  ******
1000   REM ****** INSERT Update Data SUBROUTINE HERE  ******
1500   REM ****** INSERT Start Database SUBROUTINE HERE  ******
2000   REM ****** LOAD FILE FROM DISK ******
2010  INPUT "ENTER FILE NAME :";F$
2020   OPEN "I",1,F$
2030   INPUT #1,NF
```

```
2040  FOR N=1 TO NF
2050  FOR COL=1 TO NC
2060  INPUT #1,DA$(N,COL)
2070  NEXT COL,N
2080  CLOSE
2090  RETURN
2500   REM ****** SAVE FILE TO DISK ******
2510  INPUT "ENTER FILE NAME :";F$
2520   OPEN "O",1,F$
2530   PRINT #1,NF
2540  FOR N=1 TO NF
2550  FOR COL=1 TO NC
2560  PRINT #1,DA$(N,COL);","
2570  NEXT COL,N
2580  CLOSE
2590  RETURN
2600     REM ****** CLEAR SCREEN SUBROUTINE ******
2610  CLS:PRINT:PRINT:RETURN
2620    REM ****** INKEY$ INPUT SUBROUTINE *****
2630  A$=INKEY$:IF A$="" GOTO  2630
2640  A=VAL(A$)
2650  RETURN
```

Fig. 7-1. Example of program produced by DB Starter.

The array in line 40 of the example was created and DIMensioned according to user input requirements, just as the menu was constructed and subroutines allocated for later work by the human programmer. Two subroutines relating to disk I/O were actually written entirely by the program. The finished code was then saved to disk. As written, the program will do the following things:

1) Ask the user for beginning line number, and desired line number increments.

2) Ask if a string array will be used to store data, and, if so, allow the user to specify whether the array will be one- or two-dimensional. The elements that should be DIMensioned are also input.

3) A "menu" of reasonable size (i.e., which can fit on a single screen) may be specified. Each choice can be described. Program lines to print the menu to the screen will be created, along with an "enter choice" prompt.

4) Each of the menu choices will be assigned a subroutine line number—marked with a REMark—so the programmer can flush them out later. An ON CH GOSUB line will be created, sending control to each of the menu subroutines.

5) Disk file I/O subroutines which will save or load data stored in a one- or two-dimensional array are automatically created.

6) The user can also specify several other subroutines, such as CLS:PRINT:PRINT, and A$=INKEY$: IF A$=" " GOTO.

DB Starter will, then, create the basics of a simple database management program which must be completed by the programmer. It doesn't complete the program, but does save a great deal of typing time. Arguably, there is a much simpler way of accomplishing nearly the same thing, i.e., write out an all-purpose program containing most-used modules, and then SAVE that program on a convenient disk. When the time comes to create a new program, the user can simply load the general module, delete lines not needed, renumber, and do other minor work to tailor it into a skeleton for the new project. Or structured programming techniques can be used, with common variable names, routines, etc., to build a great many program modules that can be readily transferred from one program to another.

Programs that write other programs make the most sense when developed for the unsophisticated user. That category might include someone who is incapable of taking an all-purpose program

and changing the code to fit a new purpose—a nonprogrammer, or a beginning programmer. Given a sufficiently sophisticated version of DB Starter, the user might be able to answer a series of prompts to inform the computer just what type of task had to be performed, and then receive a finished program that will do the job.

DB Starter can only do a few things. While keeping the size of the program down to what will comfortably fit in this book, I've left the door open for ambitious programmers to expand its capabilities, and to apply the concepts to their own work.

Using Figs. 7-2 and 7-3, let's look at how the program works. It consists of a series of modules, each designed to "create" a specific type of BASIC code. The mechanics are simple. The lines of the target program are assembled from the "library" of words and phrases built into DB Starter. As each line of the target program is completed, it is stored in a string array, LN$(n). The particular element of LN$(n) is determined by a counter, CU.

Each time a new target program line is initiated, control is sent

A$	Character input from keyboard through INKEY$
CFLAG	Check to see end of DATA input.
CH$	User choice input.
COL$	Number of elements in second dimension of array.
CU	Counter
D3$	Data string.
D4	Number of data items entered by user.
DI	Choice entered by user.
F$	Filename for output file.
IC	Increment for line numbers.
IOFLAG	Whether or not user will need I/O routines.
LN$(n)	Program lines being built.
MENU$	(n) Label for menu choices.
MI	Number of choices to be on menu.
N	Loop counter.
N1-N9	Loop counters.
NW	Loop counter.
P$	Substring of program line.
P1$	Substring of program line.
ROW$	Number of rows in user array.
Y$	Middle part of string.

Fig. 7-2. Variables used in DB Starter.

```
10 '      ****************************
20 '      *                          *
30 '      *   DataBase Starter       *
40 '      *                          *
50 '      ****************************
60 '
70 CLEAR 10000
80 DEFINT A-Y
90 ON ERROR GOTO 2680
100 DIM LN$(400),NU(20)
110 GOTO 140
120 A$=INKEY$:IF A$="" GOTO 120
130 RETURN
140 CU=1
150 P1$=CHR$(34)
160 P$=S2$+"PRINT"+S1$+P1$+S5$
170 CLS
180 GOSUB 340
190 GOSUB 210
200 GOTO 250
210 :  FOR N8=1 TO 3
220 :    PRINT"*";TAB(63)"*";
230 :  NEXT N8
```

```
240 RETURN

245 ' *** Instructions? ***

250 PRINT"*";TAB(17);"-- Do you want instructions ?
    --";TAB(63);"*";
260 GOSUB 210
270 PRINT"*   You may also type 'H' or 'HELP' to most input
    prompts.     *";
280 PRINT STRING$(64,"*");
290 GOSUB 120
300 IF A$="N" OR A$="n" THEN CLS: GOTO 510
310 IF A$="H" OR A$="h" THEN RUN"AUTOPROG/DOC"
320 IF A$="Y" OR A$="y" THEN RUN"AUTOPROG/DOC" ELSE 290
330 CLS
340
PRINT"*************************************************************
     *********";
350 PRINT"*
         *";
360 PRINT"*                          Automatic Programmer
          *";
```


Fig. 7-3. Program listing for DB Starter.

```
370 PRINT"*                          DB Starter
          *";
380 PRINT"*                       By:    David D. Busch
          *";
390 PRINT"*    ------------------------------------------
          ------------  *";
400 RETURN
410 CLS
420 CLOSE
430 CU=1:NU=1
440 :  FOR N=1 TO 20
450 :    NU(N)=0
460 :  NEXT N
470 NU$=""
480 :  FOR N8=1 TO 100
490 :    LN$(N8)=""
500 :  NEXT N8
510 LN=10:IC=10
520 PRINT:PRINT:PRINT
530 GOSUB 550
540 GOTO 660

545 ' *** Enter file name of program ***
```

```
550 LINE INPUT"ENTER FILE NAME  :  ";F$
560 IF LEFT$(F$, 4)="HELP" OR F$="H" OR F$="h" GOSUB 2800
570 S9=INSTR(F$,"/")
580 IF S9=0 GOTO 630
590 IF LEN(MID$(F$, S9))>4 GOTO 550
600 IF VAL(MID$(F$, S9+1, 1))>0 GOTO 550
610 IF LEN(F$)>12 GOTO 550
620 GOTO 660
630 IF LEN(F$)>8 GOTO 550
640 IF F$="" GOTO 550
650 RETURN
660 IF F$="" THEN F$="TEST"
670 GOTO 720

675 ' *** Increment line number ***

680 LN=LN+IC
690 CU=CU+1
700 LN$(CU)=STR$(LN)+"  "
710 RETURN
720 GOSUB 680

725 ' *** Start writing program ***
```

Fig. 7-3. Program listing for DB Starter. (Continued from page 63.)

```
730 LN$(CU)=LN$(CU)+"CLEAR MEM/3*2"
740 CLS:PRINT:PRINT

745 ' *** Data Lines ***

750 PRINT"Would you like to build some data lines?"
760 GOSUB 120
770 IF A$="H" OR A$="h" GOSUB 3280: GOTO 740
780 IF A$="N" OR A$="n" GOTO 1030
790 IF A$="Y" OR A$="y" GOTO 850
800 GOTO 760
810 CU=CU+1
820 GOSUB 680
830 LN$(CU)=LN$(CU)+"DATA "
840 RETURN
850 PRINT"Enter data elements to be written into program."
860 PRINT"Separate with commas.  Input no more than two lines"
870 PRINT"of DATA, then hit ENTER and input another pair of
lines."
880 PRINT "It is not necessary to enter the word DATA.  Enter
in"
890 PRINT"this form:  35,20,Address,Phone,Zip "
900 PRINT" Enter ";CHR$(34);"/";CHR$(34);" to finish."
```

```
910 LINE INPUT D3$
920 IF D3$="/" CFLAG=0: GOTO 1030
930 IF RIGHT$(D3$, 1)="/" THEN D3$=LEFT$(D3$, LEN(D3$)-1):
CFLAG=1
940 IF RIGHT$(D3$, 1)="," THEN D3$=LEFT$(D3$, LEN(D3$)-1)
950 FOR N7=1 TO LEN(D3$)
960 Y$=MID$(D3$, N7, 1)
970 IF Y$="," THEN D4=D4+1
980 NEXT N7
990 D4=D4+1
1000 GOSUB 810
1010 LN$(CU)=LN$(CU)+D3$
1020 IF CFLAG=0 GOTO 900

1025 ' *** Build arrays ***

1030 CLS:PRINT:PRINT
1040 PRINT"Will this program store disk I/O data in a string
array?"
1050 GOSUB 120
1060 IF A$="H" OR A$="h" GOSUB 2920: GOTO 1030
1070 IF A$="N" OR A$="n" THEN 1320
```

Fig. 7-3. Program listing for DB Starter. (Continued from page 65.)

```
1080 IF A$="Y" OR A$="y" THEN 1090 ELSE 1050
1090 GOSUB 680
1100 PRINT"Will the array have one or two dimensions?"
1110 GOSUB 120
1120 IF A$="H" OR A$="h" GOSUB 2920: GOTO 1100
1130 DI=VAL(A$)
1140 IF DI<1 OR DI>2 THEN 1110
1150 IF DI=1 GOTO 1230
1160 INPUT"How many elements in the first dimension (ROW)";ROW$
1170 IF LEFT$(ROW$, 1)="h" OR LEFT$(ROW$, 1)="H" GOSUB 3030:
GOTO 1160
1180 INPUT"Enter elements in second dimension (COL) :";COL$
1190 IF LEFT$(COL$, 1)="h" OR LEFT$(COL$, 1)="H"  GOSUB 3030:
GOTO 1180
1200 ROW=VAL(ROW$)
1210 COL=VAL(COL$)
1220 GOTO 1260
1230 INPUT"How large should the array be";ROW$
1240 IF LEFT$(ROW$, 1)="H" OR LEFT$(ROW$, 1)="h" GOSUB 3030:
GOTO 1230
1250 ROW=VAL(ROW$)
1260 LN$(CU)=LN$(CU)+"DIM DA$(" +STR$(ROW)
1270 IF DI=1 THEN LN$(CU)=LN$(CU)+")": GOTO 1320
```

```
1280 LN$(CU)=LN$(CU)+"," +STR$(COL)+")"
1290 GOSUB 680
1300 LN$(CU)=LN$(CU)+"NC=" +STR$(COL)
1310 IF D4>0 THEN LN$(CU-1)=LN$(CU-1)+",DTA$(" +STR$(D4)+")":
GOTO 1330
1320 IF D4>0 THEN GOSUB 680: LN$(CU)=LN$(CU)+"DIM DTA$("
+STR$(D4)+")"
1330 IF D4>0 THEN CU=CU+1: GOSUB 680: LN$(CU)=LN$(CU)+"FOR G=1
TO " +STR$(D4)+":READ DTA$(G):NEXT G

1335 ' *** Build Menus ***

1340 PRINT"Will this program need a menu?"
1350 GOSUB 120
1360 IF A$="H" OR A$="h" GOTO 2870
1370 IF A$="N" OR A$="n" THEN 1900
1380 IF A$="Y" OR A$="y" THEN PRINT A$: GOTO 1390 ELSE 1350
1390 GOSUB 680
1400 LN$(CU)=LN$(CU)+"CLS:PRINT:" +P$+"    **********  MENU
**********" +P1$+":PRINT"
1410 IM(1)=LN
1420 CLS:PRINT:PRINT
```

Fig. 7-3. Program listing for DB Starter. (Continued from page 67.)

```
1430 INPUT"How many choices on the menu";CH$
1440 IF LEFT$(CH$, 1)="H" OR LEFT$(CH$, 1)="h" GOSUB 3030: GOTO
1420
1450 MI=VAL(CH$)
1460 IF MI<2 GOTO 1420
1470 IF DI=0 THEN 1550
1480 IF MI=2 THEN CH=MI: GOTO 1580
1490 PRINT"Will the choices include 'Save file to disk' and
'Load file from disk' ? ";
1500 GOSUB 120
1510 IF A$="H" OR A$="h" GOSUB 3190: GOTO 1490
1520 IF A$="Y" OR A$="y" THEN IOFLAG=2: PRINT A$: GOTO 1550
1530 IF A$="n" OR A$="N" PRINT A$: GOTO 1550
1540 GOTO 1500
1550 CH=MI
1560 IF CH=IOFLAG THEN N=1: GOTO 1670
1570 CH=CH-IOFLAG
1580 :  FOR N=1 TO CH
1590 :    PRINT"Enter label for menu choice #";N
1600 :    INPUT MENU$(N)
1610 :    IF MENU$(N)="HELP" OR MENU$(N)="H" OR MENU$(N)="h"
GOSUB 3160: GOTO 1590
1620 :  NEXT N
```

```
1630 :  FOR NW=1 TO CH
1640 :    GOSUB 680
1650 :    LN$(CU)=LN$(CU)+P$+STR$(NW)+".)    " +MENU$(NW)+P1$
1660 :  NEXT NW
1670 IF IOFLAG=2 THEN GOSUB 680: LN$(CU)=LN$(CU)+P$+STR$(N)+".)
" +"LOAD FILE FROM DISK" +P1$: GOSUB 680:
LN$(CU)=LN$(CU)+P$+STR$(N+1)+".)  " +"SAVE FILE TO DISK" +P1$
1680 GOSUB 680
1690 LN$(CU)=LN$(CU)+"PRINT"
1700 :  FOR NW=1 TO MI
1710 :    NU=NU+IC*50
1720 :    NU(NW)=NU
1730 :    NU$=NU$+STR$(NU)+","
1740 :  NEXT NW
1750 NU$=LEFT$(NU$,(LEN(NU$)-1))
1760 GOSUB 680
1770 LN$(CU)=LN$(CU)+"INPUT" +P1$+"ENTER CHOICE : " +P1$+";CH$"
1780 GOSUB 680
1790 LN$(CU)=LN$(CU)+"CH=VAL(CH$): IF CH<1 OR CH> " +STR$(MI)+"
GOTO" +STR$(VAL(LN$(CU-1)))
1800 GOSUB 680
1810 LN$(CU)=LN$(CU)+"ON CH GOSUB" +NU$
```

Fig. 7-3. Program listing for DB Starter. (Continued from page 69.)

```
1820 GOSUB 680
1830 LN$(CU)=LN$(CU)+"GOTO " +STR$(IM(1))
1840 :  FOR N=1 TO MI-IOFLAG
1850 :    GOSUB 680
1860 :    LN=NU(N)
1870 :    LN$(CU)=STR$(NU(N))+"   REM ****** INSERT "
+MENU$(N)+" SUBROUTINE HERE " +" *******"
1880 :  NEXT N
1890 IF IOFLAG<>2 THEN 2380
1900 GOSUB 680
1910 IF DI=0 AND MI=0 GOTO 2380
1920 IF MI=0 THEN LN$(CU)=LN$(CU)+"  REM ****** LOAD FILE FROM
DISK": GOTO 1950
1930 LN=NU(N)
1940 LN$(CU)=STR$(NU(N))+"   REM ****** LOAD FILE FROM DISK
*******"
1950 GOSUB 680
1960 LN$(CU)=LN$(CU)+"INPUT " +P1$+"ENTER FILE NAME :"
+P1$+";F$"
1970 GOSUB 680
1980 LN$(CU)=LN$(CU)+" OPEN " +P1$+"I" +P1$+",1,F$"
1990 GOSUB 680
2000 LN$(CU)=LN$(CU)+" INPUT #1,NF"
```

```
2010 GOSUB 680
2020 LN$(CU)=LN$(CU)+"FOR N=1 TO NF"
2030 GOSUB 680
2040 IF DI=2 THEN LN$(CU)=LN$(CU)+"FOR COL=1 TO NC": GOSUB 680
2050 LN$(CU)=LN$(CU)+"INPUT #1,DA$(N"
2060 IF DI=2 THEN LN$(CU)=LN$(CU)+",COL)" ELSE
LN$(CU)=LN$(CU)+")"
2070 GOSUB 680
2080 LN$(CU)=LN$(CU)+"NEXT"
2090 IF DI=2 THEN LN$(CU)=LN$(CU)+" COL,N"
2100 GOSUB 680
2110 LN$(CU)=LN$(CU)+"CLOSE"
2120 GOSUB 680
2130 LN$(CU)=LN$(CU)+"RETURN"
2140 GOSUB 680
2150 IF MI=0 THEN LN$(CU)=LN$(CU)+"   REM ****** SAVE FILE TO
DISK ******": GOTO 2180
2160 LN=NU(N+1)
2170 LN$(CU)=STR$(NU(N+1))+"   REM ****** SAVE FILE TO DISK
******"
2180 GOSUB 680
2190 LN$(CU)=LN$(CU)+"INPUT " +P1$+"ENTER FILE NAME :"
+P1$+";F$"
```


Fig. 7-3. Program listing for DB Starter. (Continued from page 71.)

```
2200 GOSUB 680
2210 LN$(CU)=LN$(CU)+" OPEN " +P1$+"O" +P1$+",1,F$"
2220 GOSUB 680
2230 LN$(CU)=LN$(CU)+" PRINT #1,NF"
2240 GOSUB 680
2250 LN$(CU)=LN$(CU)+"FOR N=1 TO NF"
2260 GOSUB 680
2270 IF DI=2 THEN LN$(CU)=LN$(CU)+"FOR COL=1 TO NC": GOSUB 680
2280 LN$(CU)=LN$(CU)+"PRINT #1,DA$(N"
2290 IF DI=2 THEN LN$(CU)=LN$(CU)+",COL)" ELSE
LN$(CU)=LN$(CU)+")"
2300 LN$(CU)=LN$(CU)+";" +P1$+"," +P1$
2310 GOSUB 680
2320 LN$(CU)=LN$(CU)+"NEXT"
2330 IF DI=2 THEN LN$(CU)=LN$(CU)+" COL,N"
2340 GOSUB 680
2350 LN$(CU)=LN$(CU)+"CLOSE"
2360 GOSUB 680
2370 LN$(CU)=LN$(CU)+"RETURN"

2375 ' *** Subroutines ? ***

2380 PRINT"Do you want a 'CLEAR SCREEN' subroutine?  ";
```

```
2390 GOSUB 120
2400 IF A$="H" OR A$="h" GOSUB 3250: GOTO 2380
2410 IF A$="Y" OR A$="y" PRINT A$: GOTO 2440
2420 IF A$="N" OR A$="n" PRINT A$: GOTO 2480
2430 GOTO 2390
2440 GOSUB 680
2450 LN$(CU)=LN$(CU)+"   REM ****** CLEAR SCREEN SUBROUTINE
******"
2460 GOSUB 680
2470 LN$(CU)=LN$(CU)+"CLS:PRINT:PRINT:RETURN"
2480 PRINT"Do you want an 'INKEY$-INPUT' subroutine?  ";
2490 GOSUB 120
2500 IF A$="H" OR A$="h" GOSUB 3250: GOTO 2480
2510 IF A$="N" OR A$="n" PRINT A$: GOTO 2620
2520 IF A$="Y" OR A$="y" PRINT A$: GOTO 2540
2530 GOTO 2490
2540 GOSUB 680
2550 LN$(CU)=LN$(CU)+"  REM ****** INKEY$ INPUT SUBROUTINE
*****"
2560 GOSUB 680
2570 LN$(CU)=LN$(CU)+"A$=INKEY$:IF A$=" +P1$+P1$+" GOTO "
+STR$(LN)
```


Fig. 7-3. Program listing for DB Starter. (Continued from page 73.)

```
2580 GOSUB 680
2590 LN$(CU)=LN$(CU)+"A=VAL(A$)"
2600 GOSUB 680
2610 LN$(CU)=LN$(CU)+"RETURN"

2615 ' *** Write program to disk ***

2620 OPEN"O",1, F$
2630 :  FOR N1=1 TO CU
2640 :    PRINT#1, LN$(N1)
2650 :  NEXT N1
2660 CLOSE
2670 RUN

2675  ' *** Error Trap ***

2680 PRINT:PRINT
2690 PRINT TAB(20)"*****  UNKNOWN ERROR  *****"
2700 PRINT TAB(25)"IN LINE ";ERL
2710 FOR N9=1 TO 500
2720 NEXT N9
2730 STOP
2740 CLS:PRINT:PRINT
```

```
2750 RETURN
2760 PRINT
2770 PRINT TAB(15)"Hit any key to resume program"
2780 GOSUB 120
2790 RETURN
2795 ' *** Help Routines ***
2800 GOSUB 2740
2810 PRINT"You should enter the filename you want  -- it must"
2820 PRINT"be a legal Disk basic name, or your input will be"
2830 PRINT"rejected."
2840 PRINT
2850 LINE INPUT"ENTER FILENAME :";F$
2860 RETURN
2870 GOSUB 2740
2880 PRINT"Menus may be designed using a special"
2890 PRINT"module that asks for number of choices, labels, etc."
2900 GOTO 1340
2910 GOTO 2760
2920 GOSUB 2740
2930 PRINT"Many forms of data are conveniently stored in a
string"
```


Fig. 7-3. Program listing for DB Starter. (Continued from page 75.)

```
2940 PRINT"array which looks like this: DA$(row,col).  A
checkbook"
2950 PRINT"represents data that can be stored in a
two-dimensional"
2960 PRINT"array.  Each check number represents a row, while
payee"
2970 PRINT"amount, balance, etc. represent columns.  These
arrays"
2980 PRINT"can be conveniently stored and loaded to and from
disk."
2990 PRINT"Use a one-dimensional  array for information which
has only"
3000 PRINT"one 'field' per record.  If rows and columns are
involved"
3010 PRINT"use a two dimensional array."
3020 GOTO 2760
3030 GOSUB 2740
3040 PRINT"Enter how large each dimension of the array should
be"
3050 PRINT"For example, might want an array: DA$(30,30)."
3060 PRINT"Do not make much larger than you need to save
memory."
3070 GOTO 2760
```

```
3080 GOSUB 2740
3090 PRINT"Most programs with multiple functions need a menu so
the"
3100 PRINT"user may choose.  Automatic Programmer will design a
menu"
3110 PRINT"for you and write appropriate  input and error
trapping"
3120 PRINT"routines.  Or, you may design your own menu.  You
must"
3130 PRINT"then write your own input routine, or use the INKEY$"
3140 PRINT"subroutine provided."
3150 GOTO 2760
3160 GOSUB 2740
3170 PRINT"Enter the label or prompt for this menu choice :"
3180 GOTO 2760
3190 GOSUB 2740
3200 PRINT"If you have specified a string array, and need disk
I/O"
3210 PRINT"You should enter Yes.  Program will write these
routines"
3220 PRINT"for you, and reduce number of menu choices you have"
```

Fig. 7-3. Program listing for DB Starter. (Continued from page 77.)

```
3230 PRINT"to input by two.  Menu labels will be created for
you."
3240 GOTO 2760
3250 GOSUB 2740
3260 PRINT"Enter Yes if you want this subroutine in your
program"
3270 GOTO 2760
3280 GOSUB 2740
3290 PRINT"You may build data lines automatically, along with"
3300 PRINT"a routine to read them into a string array.  Just"
3310 PRINT"enter the data information when asked"
3320 GOTO 2760
```

Fig. 7-3. Program listing for DB Starter. (Continued from page 79.)

to a subroutine at line 680. There, the line number of the target (LN) is incremented by IC (LN=LN+IC) where IC is specified by the user. Next, CU is increased by one so that the new program line will be stored in the next available element of LN$(n). Finally, the new line number (LN) is converted into a string, and assigned as the first part of LN$(CU), along with a pair of spaces.

For example, if LN=100 and IC=10 when control is sent to Line 680 of DB Starter, LN$(CU) will equal "110 " when it RETURNS. Each element of LN$(n) will begin with a line number, usually larger by IC from the previous element. The exception is when LN has been given a different value somewhere else in the program.

The initial line of the program will CLEAR two thirds of memory. Next, DB Starter asks the user whether or not a string array will be used to store data. If so, the number of dimensions are input into variable DI. If DI=2, then the user is asked to provide the desired size for each of the two dimensions (ROW and COL). If DI=1, then only ROW is used. The target program line is created by combining the line number (already stored in LN$(n)) with DIM and the array dimensions enclosed in parentheses. If a two-dimensional array has been specified, an additional line is developed that defines variable NC (number of columns) equal to COL. NC is used later in the target program to control disk input and output.

If menu is needed, DB Starter obligingly creates a line that labels one. Note that to make a PRINT statement it is necessary to combine PRINT with quotes around the material to be printed. Quotation marks (CHR$(34)) are stored in P1$, and this string variable used whenever quotation marks are desired in the target program.

The user is asked to input the number of choices required for the menu. If DI=0 (that is, no string array was dimensioned), the program assumes that disk file I/O will not be required, and does not offer the choice of taking advantage of the built-in disk I/O subroutines. Of course, disk files consisting of nothing but numeric values are possible, but the greater flexibility of storing both string and numeric data as strings (and then converting to numbers with VAL, as needed) makes it simpler for DB Starter to assume that disk files will be loaded into and out of a string array only.

If a string array has been specified, then the user is asked if "save file to disk" and "load file from disk" will be included in the menu. If so, IOFLAG is set to 2. The user has told the program how many choices will be included on the menu. This value is trans-

ferred to CH, which is used as a parameter in a FOR-NEXT loop that allows input of the names of the menu choices.

If the built-in disk I/O routines are desired, two is subtracted from CH so that the user does not have to bother to input these. That is, if five menu choices will be used, but two of them will be for disk I/O, the programmer has to enter only the other three. Then, the menu display lines are created for all but the disk routines.

Now things begin to get a little tricky. For each menu choice the program has to create a subroutine location to which the target program can branch, and space has to be allocated for them. So, rather than using LN and incrementing it by IC, another variable, NU, is used instead. NU is incremented by IC*50 for each of the menu subroutines. For example, if IC=10, then each of the subroutines will be spaced 500 lines apart from each other. The starting line number for each menu subroutine is stored in an array NU(n).

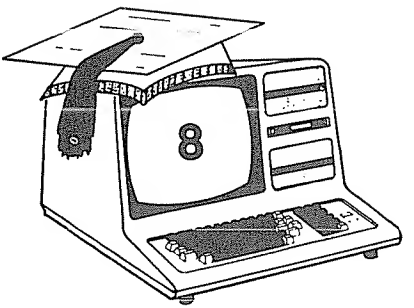
Next, a string representation of the starting line number for each menu subroutine is needed (for an ON CH GOSUB 500, 1000,1500, etc., statement). These are assembled with a comma tacked on the end. Next, an INPUT "ENTER CHOICE :";CH$ line is created for the target program and an error trap is also built. When the target program is run, if VAL(CH$) is less than one or is greater than MI (the number of menu choices available), the input is refused.

All these subroutines targeted in the ON CH GOSUB line will eventually RETURN, so control is sent back to the beginning of the menu. Its starting line number had been stored earlier in IM(1) and is used in line 670 to build a control-branching instruction. To aid the programmer in finishing the skeletal program, a REM is inserted at each of the menu subroutine starting line numbers. Remember that it's not a good idea to send control to a REM line (these might be deleted later), so don't simply begin writing the code at the next available line number following the remark.

The next portion builds a simple disk input module, which will ask the user a filename, open that sequential file, input from the file the number of items in the file, and then begin a FOR-NEXT loop from 1 to the number of items in the file. Within the loop, INPUT #1 loads the data. If the relevant array is two-dimensional, a nested FOR-NEXT loop from 1 to the number of columns (NC, defined early in the program) is used. Actual construction of the disk input module is fairly clear cut. Its mirror-image twin is the Create Disk Output routine, which performs its own function in nearly identical fashion.

Other frequently needed modules can be added to DB Starter's "library" as required. I used "clear screen" and INKEY$ routines as examples; you are free to add your own favorite subroutines as you desire. The final portion of the program saves the finished target program to disk under any desired legal name. A noncompressed (ASCII) file is created which may be loaded, finished, debugged, and used as desired.

DB Starter is simple enough to form the basis for a much more complex code-generating system. A big drawback is the need to anticipate exactly what capabilities will be needed in the finished program. If a subroutine isn't in the program generating system's library, or if the parameters are beyond its capabilities (i.e., a three-dimensional array is required), the necessary code will have to be built up from scratch. It is still beyond the capability of microcomputers to use logic to create. Our silent servants must wait for instructions from us before doing anything at all, no matter how simple.



Program Proofer

In the two previous "Automatic Programmer" examples, we've shown you how to let your computer write its own screens and assemble program skeletons. Now here's "Program Proofer," which allows a TRS-80 to partially debug its own programs by checking the spelling of keywords and some syntax errors.

Some program errors caused by misspelled words lurk deep within seldom-called code. Obvious bugs will ordinarily surface during program development, because the interpreter will note a syntax error when the line is run. Other errors, however, will not be detected for some time, because the specific conditions that invoke that program line are rare. In the worst possible situations, these mistakes are hidden in error traps designed to help the unsophisticated user, or they may cause the loss of valuable data. Program Proofer will check every line of a program, and detect all bad keywords. It will catch only typos, however; if you used LPRINT when you meant PRINT, the bug will slip by unchecked.

Program Proofer was inspired by the plethora of spelling checker programs which have become available in the past few years. These useful software tools take any text document and compare each word to an internal dictionary. Any word in your text which does not appear in the dictionary is flagged as a possible spelling error. This program works on exactly the same principal, but with a much smaller dictionary of 112 keywords, most of which

are those used in Radio Shack's Level II, Model III, or Model 4 BASIC interpreters.

Program Proofer examines every word in a target program, ignoring words inside quotes—prompts, for example—numbers, and arithmetic operators. The only letter combinations left are keywords, variables, and misspelled words. Although it would be possible to tell which of the remaining words are variables, leaving only the incorrect keywords, I decided not to implement this feature. As written, Program Proofer has the added capability of providing a variable cross-reference listing that includes line numbers.

Not throwing out variables also means that the operator has the opportunity to look for variables which may have been spelled incorrectly as well. This is important both to Model I/III users, as well as Model 4 users, but for different reasons. Under Level II or Model III BASIC, PREVIOUS and PEVIOUS would appear as two different variables, although PREVIOUS and PREVIUS would not. With the earlier Microsoft BASIC, only the first two letters of the variable name are significant, so finding such misspellings is important. With the Model 4, however, longer variable names are allowed, so finding errors is even more important. PREVIOUS and PREVIUS would, in fact, be different variables and cause an error if the difference were unintentional.

The target program should not be one which has been tightly packed with all spaces removed. These spaces are required with Model 4's BASIC, but optional under Level II or Model III BASIC. Multiple statements per line are okay. However, keywords should have spaces separating them, and there should be a space after the line number and before the first word in the line. Other spaces may be omitted. If you wish to proof a program which has been tightly packed, use a utility such as PACKER, from Cottage Software. This is an indispensable programmer's tool that is especially helpful for deciphering someone else's coding logic.

When asked for the target program name, enter the file specification of the previously saved ASCII format program. Each line will be examined separately, and all words not included within quotation marks compared with the internal dictionary. If a match is not found, the questionable word (which may also be a variable) is stored away for later reference. The number of parentheses are counted, and any missing ones noted. Program Proofer will also locate absent quotation marks, and list all the variables used in the program. For those who are using NEWDOS/80 2.0, the bad words and variables are

presented in sorted, alphabetical order. In all cases, line numbers are provided to make tracking down the errant bugs easier.

Here, briefly, is how Program Proofer works. (Refer to Figs. 8-1 and 8-2.) The 112 keywords are stored in a string array, WRD$(26,16). Each of 26 rows in the array correspond to one of the 26 letters of the alphabet. The 16 columns allow for up to 16 keywords beginning with that letter. For example, ABS is stored in WRD$(1,1), while AND is placed in WRD$(1,2).

This is accomplished in a FOR-NEXT loop beginning at line 760. The keyword is read from a data line, and the first letter examined to determine its ASCII value. Then 64 is subtracted from that value to obtain the alphabetic position, and thus the corresponding ROW of WRD$(row,col). The keyword CDBL, which begins with C (ASCII 67), is directed to Row 3 (67 minus 64). The column is determined by a counter, A, which is incremented every time a new keyword is READ, and reset to one each time a new ROW is opened (A2 < > PREVIOUS).

A$	Line of text being proofed.
BAD$(n)	Array storing bad words and variables.
D$	Temporarily stores good keyword names.
D2	ASC value of first character in keyword.
DOSFLAG	Set to 1 if NEWDOS/80 used.
F$	Filename of program being proofed.
L	Length of the program segment being proofed.
LP	Number of left parentheses.
M$	Middle string of SEG$
N	Loop counter.
N1-N9	Loop counters.
NI, NU	Counters
P	Position of space in program line being checked.
PAR$(n)	Lines with odd number of parentheses.
PFLAG	Send output to printer.
RP	Number of right parentheses.
SEG$	Program segment being proofed.
TEST$	Program segment being tested.
WRD$(n,n1)	Array storing good keywords.
Z3	Number of lines printed.
ZU	Number of lines printed.

Fig. 8-1. Variables used in Program Proofer.

As disk operating systems gain new features in their BASICs, Program Proofer may be updated to include these new keywords and commands. Add the word to the proper position in the DATA lines and change the 112 to the new number of keywords. If a given letter of the alphabet now has more than 16 keywords, it will be necessary to reDIMension WRD$(row,col) as well.

The target program (F$) is OPENed, and LINEINPUT into variable A$, a line at a time. The first space in the program line is assumed to follow the line number, and the rest of the line is stored in SEG$. A FOR-NEXT loop from 1 to L+1 (length of SEG$) examines each character in the program line in turn.

When certain delimiters are reached, the program assumes that the end of a word or variable has been located. These delimiters include a space, quotation mark, comma, semicolon, parentheses, colon, and arithmetic signs such as plus, minus, equals, more than, or less than. At this point, control drops to a subroutine, where that portion of the line, TEST$, is subjected to a series of tests.

If TEST$=" " (null), or if the value of the first character is greater than zero (signifying a number), then the program jumps back and begins looking at the next section of the program line. Obviously, no variable or keyword can begin with a number. When "REM" or its abbreviation " ' " is encountered, the program knows that the rest of the program line should be ignored.

Once TEST$ passes these checks, it enters a FOR-NEXT loop from 1 to 16, which compares TEST$ with all the elements of WRD$(row,col) beginning with the same letter of the alphabet as TEST$. If a match is found, FLAG is set to 1 and control drops to 1230, where counter NU is incremented and the suspect word stored in string array BAD$(n), along with the line number where it appears. The word itself is positioned first, followed by the line number, so that the array may later be sorted into alphabetical order. Finally, TEST$ is nulled and the rest of the line examined for additional statements, variables, and keywords.

Any time a quotation mark is encountered, SFLAG is set to 1, and additional characters in the line are ignored until the second ("close quote") is located. Then the following words are considered and checked normally. Though no specific check for missing quotation marks is built in, they will stand out like a sore thumb because, in the final listing, words inside of prompts will be listed as bad words.

A check is included for absent parentheses, however. Each right parenthesis encountered in a program line increments variable

```
10 '      ****************************
20 '      *                          *
30 '      *    Program Proofer       *
40 '      *                          *
50 '      ****************************
60 '
70 CLEAR 10000
80 DEFINT A-Y
90 ON ERROR GOTO 1720
100 DIM WRD$(26, 16), PAR$(30), BAD$(200)
110 GOTO 140
120 A$=INKEY$:IF A$="" GOTO 120
130 RETURN
140 CLS
150 GOSUB 310
160 GOSUB 180
170 GOTO 220
180 :  FOR N8=1 TO 3
190 :    PRINT"*";TAB(63)"*";
200 :  NEXT N8
210 RETURN

215 ' *** Instructions ***
```

```
220 PRINT"*";TAB(17);"-- Do you want instructions ?
    --";TAB(63);"*";
230 GOSUB 180
240 PRINT"*   You may also type 'H' or 'HELP' to most input
    prompts.     *";
250 PRINT STRING$(64,"*");
260 GOSUB 120
270 IF A$="N" OR A$="n" THEN CLS: GOTO 400
280 IF A$="H" OR A$="h" THEN RUN"AUTOPROG/DOC"
290 IF A$="Y" OR A$="y" THEN RUN"AUTOPROG/DOC" ELSE 260
300 CLS
310
PRINT"**********************************************************
     ********";
320 PRINT"*
          *";
330 PRINT"*                    Automatic Programmer
       *";
340 PRINT"*                     PROGRAM PROOFER
       *";
350 PRINT"*                   By:    David D. Busch
       *";
```

Fig. 8-2. Listing for Program Proofer.

```
360 PRINT"*   ------------------------------------------------
          ------------  *";
370 RETURN
380 CLS
390 CLOSE
400 PRINT:PRINT:PRINT
410 GOSUB 430
420 GOTO 540

425 ' *** Input filename to be proofed ***

430 LINE INPUT"ENTER FILE NAME  :  ";F$
440 IF LEFT$(F$, 4)="HELP" OR F$="H" OR F$="h" GOSUB 1920
450 S9=INSTR(F$,"/")
460 IF S9=0 GOTO 510
470 IF LEN(MID$(F$, S9))>4 GOTO 430
480 IF VAL(MID$(F$, S9+1, 1))>0 GOTO 430
490 IF LEN(F$)>12 GOTO 430
500 GOTO 540
510 IF LEN(F$)>8 GOTO 430
520 IF F$="" GOTO 430
530 RETURN
540 IF F$="" THEN F$="TEST"
```

```
550 RESTORE
560 DATA ABS, AND, ASC, ATN, BASIC, BOOT
570 DATA CDBL, CHR$, CINT, CHAIN, CLEAR, CLOSE, CLOCK, CLS,
    CMD, COS, COPY, CSNG, CVD, CVI, CVS, DEFFN, DATA, DATE,
    DIR, DEFDBL, DEFINT, DEFSNG, DEFSTR, DIM
580 DATA ELSE, END, EOF, ERL, ERR, ERROR, EXP, FIELD, FOR,
    FORMAT, FN, FRE$, FREE, FIX, GOTO, GOSUB, GET
590 DATA IF, INP, INPUT, INKEY$, INSTR, INT, KILL, LET, LPRINT,
    LINEINPUT, LSET, LOAD, LEN, LEFT$
600 DATA LOG, MEM, MID$, MEM, MERGE, MKD$, MKI$, MKS$
610 DATA NEW, NEXT, NOT, ON, OR, OPEN, OUT, PEEK, PRINT, POINT,
    POKE, POS, PUT
620 DATA RANDOM, RIGHT$, READ, REM, RESET, RESTORE, RESUME,
    RETURN, RND, RUN
630 DATA SAVE, SET, SGN, SIN, SQR, STEP, STOP, STR$, STRING$
640 DATA TAB, TAN, THEN, TIME$, TO, TROFF, TRON, USING, USR,
    VAL, VARPTR
650 CLS:PRINT:PRINT
660 PRINT TAB(10)"THIS MODULE WORKS ONLY ON FILES WHICH HAVE"
670 PRINT TAB(10)"BEEN SAVED IN NON-COMPRESSED (ASCII) FORMAT"
680 PRINT TAB(10) " Use this syntax:
    SAVE ";CHR$(34);"filename";CHR$(34)",A"
```


Fig. 8-2. Listing for Program Proofer. (Continued from page 89.)

```
690 PRINT
700 PRINT "May not work on 'packed' files, or those with  less"
710 PRINT"than one space between line number and first
    statement"
720 PRINT"on program line.  If you see garbage loading, you
    have"
730 PRINT"forgotten to save your file in ASCII format."
740 PRINT
750 PRINT @ 718," -- A  few seconds please -- "
755 ' *** Read GOOD names into array ***
760 :  FOR N=1 TO 112
770 :    READ D$
780 :    D2=ASC(LEFT$(D$, 1))-64
790 :    IF D2<>PREVIOUS THEN PREVIOUS=D2: D=1
800 :    WRD$(D2, D)=D$
810 :    D=D+1
820 :  NEXT N
830 PRINT @ 718,"Are you running Newdos 80 2.0 ?          ";
840 GOSUB 120
850 IF A$="H" OR A$="h" GOSUB 2000: CLS: GOTO 830
860 IF A$="N" OR A$="n" THEN DOSFLAG=1
870 PRINT:PRINT
```

```
880 CLS:PRINT:PRINT
890 PRINT TAB(14)" --   Reading in Program Lines -- "
900 PRINT

905 ' *** Open Program, Read in Lines ***

910 OPEN"I",1, F$
920 IF EOF(1)THEN 1330
930 LINE INPUT#1, A$
940 TEST$=""
950 PRINT A$
960 FL=0
970 SFLAG=0
980 P=INSTR(A$, CHR$(32))
990 SEG$=MID$(A$, P+1)
1000 L=LEN(SEG$)+1

1005 ' *** Check for keyword delimiter ***

1010 :  FOR N1=1 TO L
1020 :    M$=MID$(SEG$, N1, 1)
1030 :    IF SFLAG<>1 THEN 1050
```

Fig. 8-2. Listing for Program Proofer. (Continued from page 91.)

```
1040 :    IF M$=CHR$(34)THEN 1080 ELSE 1250
1050 :    IF M$=")" OR M$="+" OR M$="-" OR M$=CHR$(32)OR
          M$="="OR M$="(" OR M$=CHR$(34)OR M$="," OR M$=":" OR
          M$="<"OR M$=">" OR M$="#" OR M$="/" OR M$="*" OR
          M$=CHR$(10)OR M$="" THEN 1080
1060 :    TEST$=TEST$+M$
1070 :    GOTO 1250
1080 :    IF SFLAG=1 THEN SFLAG=0: TEST$="": GOTO 1250
1090 :    IF M$=CHR$(34)THEN SFLAG=1:IF MID$(SEG$, N1-1,
          1)=CHR$(32)THEN TEST$=""
1100 :    IF M$="(" THEN LP=LP+1
1110 :    IF M$=")" THEN RP=RP+1
1120 :    FL=0
1130 :    IF TEST$="" THEN 1250
1140 :    IF TEST$="REM" OR TEST$="'" THEN 1260
1150 :    IF VAL(TEST$)>0 THEN TEST$="": GOTO 1250
1160 :    A=ASC(LEFT$(TEST$, 1))
1170 :    IF A<65 THEN TEST$="": GOTO 1250
1180 :    A=A-64
1190 :      FOR N2=1 TO 16
1200 :        IF WRD$(A, N2)="" THEN N2=16: GOTO 1230
1210 :        IF TEST$=WRD$(A, N2)THEN FLAG=1: N2=16: GOTO 1230
1220 :      NEXT N2
```

```
1230 :     IF FLAG=0 THEN NU=NU+1: BAD$(NU)=TEST$+" : LINE "
           +LEFT$(A$, P)
1240 :     TEST$=""
1250 :  NEXT N1
1260 IF RP=LP THEN 1310
1270 NI=NI+1

1275 ' *** Paren missing ***

1280 PAR$(NI)="LINE " +LEFT$(A$, P)+" :  MISSING "
1290 IF RP>LP THEN P$="LEFT" ELSE P$="RIGHT"
1300 PAR$(NI)=PAR$(NI)+P$+" PARENTHESIS"
1310 RP=0:LP=0
1320 GOTO 920

1232 ' *** Display results ***

1330 CLS:PRINT:PRINT
1340 PRINT"Do you want output to go to printer?"
1350 GOSUB 120
1360 IF A$="Y" OR A$="y" THEN PFLAG=1
1370 GOSUB 1640
```


Fig. 8-2. Listing for Program Proofer. (Continued from page 93.)

```
1380 IF DOSFLAG=0 GOSUB 1700
1390 ZU=1

1395 ' *** Show BAD words and Variables ***

1400 :  FOR N4=1 TO NU
1410 :    IF ZU/11=INT(ZU/11)GOSUB 1600
1420 :    IF BAD$(N4)=BAD$(N4-1)THEN 1460
1430 :    PRINT BAD$(N4)
1440 :    IF PFLAG=1 THEN LPRINT BAD$(N4)
1450 :    ZU=ZU+1
1460 :  NEXT N4
1470 GOSUB 1600
1480 Z3=1

1485 ' *** Show Missing Parens ***

1490 :  FOR Z3=1 TO NI
1500 :    IF Z3/11=INT(Z3/11)GOSUB 1600
1510 :    PRINT PAR$(Z3)
1520 :    IF PFLAG=1 LPRINT PAR$(Z3)
1530 :  NEXT Z3
1540 PRINT
```

```
1550 PRINT TAB(20)" -- END OF LIST -- "
1560 PRINT
1570 PRINT TAB(15)"HIT ANY KEY TO RETURN TO MAIN MENU"
1580 GOSUB 120
1590 GOTO 380
1600 PRINT
1610 PRINT TAB(22)"HIT ANY KEY"
1620 GOSUB 120
1630 RETURN
1640 CLS:PRINT:PRINT
1650 PRINT
1660 PRINT
1670 PRINT TAB(14)"  ** POSSIBLE MISPELLINGS AND VARIABLES **"
1680 PRINT
1690 RETURN
1700 CMD"O",NU, BAD$(1)
1710 RETURN

1715 ' *** Error Trap ***

1720 IF ERR/2+1<>54 GOTO 1790
1730 CLS:PRINT
```

Fig. 8-2. Listing for Program Proofer. (Continued from page 95.)

```
1740 PRINT TAB(20)"That file does not exist!"
1750 FOR N9=1 TO 500
1760 NEXT N9
1770 CLS
1780 RESUME 870
1790 PRINT:PRINT
1800 PRINT TAB(20)"*****  UNKNOWN ERROR  *****"
1810 PRINT TAB(25)"IN LINE ";ERL
1820 FOR N9=1 TO 500
1830 NEXT N9
1840 RESUME 380
1850 CLS:PRINT:PRINT
1860 RETURN
1870 PRINT
1880 PRINT TAB(15)"Hit any key to resume program"
1890 GOSUB 120
1900 RETURN
1910 GOSUB 1850

1915 ' *** Help Routine ***

1920 CLS:PRINT
```

```
1930 PRINT"Program wants the name of file to be proofread.
     Must"
1940 PRINT"be a legal Disk basic name, or your input will be"
1950 PRINT"rejected."
1960 PRINT
1970 LINE INPUT"ENTER FILENAME :";F$
1980 RETURN
1990 GOSUB 1850
2000 CLS:PRINT
2010 PRINT"If you have Newdos 80 2.0 or a later release"
2020 PRINT"with the CMD";CHR$(34);"O";CHR$(34);"sorting
     feature"
2030 PRINT"this program will sort your bad words and variables
     for you."
2040 GOTO 1870
2050 GOSUB 1850
```


Fig. 8-2. Listing for Program Proofer. (Continued from page 97.)

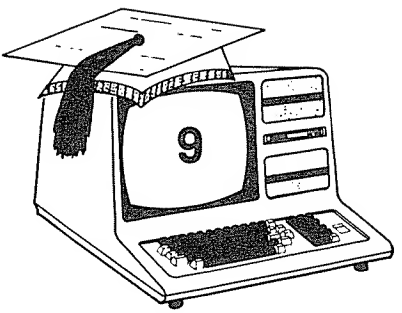
RP, while left parenthesis increase the value of LP. After the whole program line has been checked, Program Proofer compares LP and RP. If they don't match, then the line in which the error appears is stored in a string array PAR$(n), along with a note as to whether a left or right parenthesis is missing. (If one statement is missing a left parenthesis, while another statement later in that line is missing a right parenthesis, the values of LP and RP will match and the error will not be caught. This should, however, be very rare.

When the end of file (EOF) marker is encountered, the user is asked if results should be directed to a printer as well as to the screen. Then, if NEWDOS/80 2.0 is available, the array BAD$(n) is sorted into alphabetical order. Those of you using other disk operating systems or sorting routines should note that taking advantage of this feature requires only a single line; the CMD"O" invokes the sort. "NU" is specified to indicate that all NU units of the array BAD$(n) should be sorted, beginning with element number 1. If for some reason you do not want the array sorted, even though NEWDOS/80 2.0 is available, simply "lie" to the program when asked.

The suspect words are then printed out in groups of 11 words/lines. A counter, CU, keeps track of how many words are printed or listed. A word/line combination is displayed only if it does not equal the previous word/line, so if a variable or bad word appears several times in a single line, it is pointed out just once. When CU can be evenly divided by 11, the program branches to a "paging" subroutine at line 1010. Once the variables and bad words are listed, the program displays all the lines which contain missing parentheses.

A number of enhancements are possible. The program could be extended to check each variable against the keyword list, using INSTR, to see if any have inadvertently included an illegal keyword. This would be especially helpful for those of you with Model I/III computers who like to use long, descriptive variable names.

Checking the spelling of a computer program is much easier than proofreading a document, because the number of legal words is severely limited. Once a computer is told what words are allowable in a program, it is a simple matter to leave some of the tedious debugging to the machine.



Automatic Programmer Documentation

Care to coast awhile? Here's the program you don't even have to key in. Well, that is not entirely accurate. "Automatic Programmer Documentation" is a help file for the preceding three modules. It is included here to demonstrate how such help programs can be used to make a complex piece of software more usable by a beginner. The program itself actually has no other function than to serve as an introduction to the Automatic Programmer series. You have four options in this case.

1) If you have purchased the disk containing all the programs in this book, the program should be included on your disk running the three Automatic Programmer programs. It will be called as needed, and serve as a menu gateway to the others.

2) You may type in the program as presented.

3) You can type in the BASIC program lines, but write the others using Screen Editor. It will prepare the screens for you with less typing on your part.

4) Just skip this chapter entirely and do without the help file when running the other three programs.

A$	User input from keyboard through INKEY$
L$	String of 64 asterisks.
N	Loop counter.

Fig. 9-1. Variables used in Autoprogrammer Instructions.

```
10 ' ************************************
20 ' *                                  *
30 ' *  Auto Programmer Instructions    *
40 ' *                                  *
50 ' ************************************
60 CLEAR 5000
70 ON ERROR GOTO 270
80 L$=STRING$(64,"*")
90 CLS
100
PRINT"*************************************************************
     ********";
110 PRINT"*                        Automatic Programmer
        *";
120 PRINT"*
        *";
130 PRINT"*                      By:    David D. Busch
        *";
140 PRINT"*   ------------------------------------------
    ------------  *";
150 PRINT"*      This program allows  you to use your computer
    to write  *";
160 PRINT"*   some of the Basic program lines for many common
```

```
     programs    *";
170 PRINT"*    automatically.    It will produce a 'skeleton'
    coding       *";
180 PRINT"*    structure which you can 'flesh' out with
    subroutines of     *";
190 PRINT"*    your own.    Many initial 'housekeeping' tasks,
    such as      *";
200 PRINT"*    dimensioning an array, CLEARing memory, writing
    instruct-   *";
210 PRINT"*    ional screens (like this one), menus, are done
    for you.     *";
220 PRINT"*";TAB(63)"*";
230 PRINT "*                          --  HIT ANY KEY  --
    *";
240 PRINT L$;
250 IFINKEY$=""THEN250
260 GOTO 310
270 IF ERR/2+1=54 GOTO 290
280 PRINT "UNKNOWN ERROR IN LINE #";ERL:FOR N=1 TO 500:NEXT
N:RESUME 90
290 PRINT"PLEASE INSERT DISK CONTAINING AUTOMATIC"
300 PRINT"PROGRAMMER IN DISK DRIVE":RESUME 1110
```


Fig. 9-2. Program listing for Autoprogrammer Instructions.

```
310 CLS:PRINT "**********************************************
    **********************";
320 PRINT "*
          *";
330 PRINT "*        Data base management programs lend
    themselves  to       *";
340 PRINT "*    this approach.   Automatic Programmer has a
    number of        *";
350 PRINT "*    useful functions that will save you time:
          *";
360 PRINT "*
          *";
370 PRINT "*        1.) You can use it to write instructional
    screens.       *";
380 PRINT "*    Instead of mapping out pages, like this one, and
    writing    *";
390 PRINT "*    program lines to reproduce the text on the
    screen, you      *";
400 PRINT "*    can enter the material exactly as you want it to
    appear     *";
410 PRINT "*    using cursor control and full-screen editing.
    All alpha  *";
420 PRINT "*    numeric characters and symbols may be used.
    Then, pro-    *";
```

```
430 PRINT "*   gram lines will be written and saved to disk.
          *";
440 PRINT "*                          --  HIT ANY KEY  --
          *";
450 PRINT L$;
460 IF INKEY$="" GOTO 460
470 CLS:PRINT
"*****************************************************
    ***********";
480 PRINT "*       After a screen has been created, you may
    renumber it   *";
490 PRINT "*   so that the line numbers do not conflict with an
     existing  *";
500 PRINT "*   program, and MERGE the two.  This process may be
   repeated  *";
510 PRINT "*   to create several frames or menus for a
    business, computer *";
520 PRINT "*   aided instruction, games, or other program.
          *";
530 PRINT "*
         *";
540 PRINT "*      2.) Automatic Programmer may also be used to
    create     *";
```


Fig. 9-2. Program listing for Autoprogrammer Instructions. (Continued from page 103.)

```
550 PRINT "*   entire program skeletons for you to work with.
    The        *";
560 PRINT "*   'screen' writer module may be used, along with
    several     *";
570 PRINT "*   others.  It will write program lines to
    dimension  a       *";
580 PRINT "*   string array, build disk I/O routines to fill an
    array     *";
590 PRINT "*   and dump its contents to a disk file.
          *";
600 PRINT "*                          --  HIT ANY KEY  --
          *";
610 PRINT L$;
620 IF INKEY$=""GOTO 620
630 CLS:PRINT L$;
640 PRINT "*        If your program will use DATA lines, you may
    simply    *";
650 PRINT "*   enter the actual data itself.  Automatic
    Programmer will   *";
660 PRINT "*   insert line numbers, DATA statements, and write
    a routine  *";
670 PRINT "*   to READ that data into an array for later
    manipulation.    *";
```

```
680 PRINT "*       You may construct a menu, too.  If you
    choose   to     *";
690 PRINT "*   build a custom menu, you can make use  of  the
    screen      *";
700 PRINT "*   writer routine.  The computer can also build a
    menu for     *";
710 PRINT "*   you, from your input of the number of choices,
    labels for   *";
720 PRINT "*   those choices, and other data.
    *";
730 PRINT "*       When using this feature, the program will
    write ON...  *";
740 PRINT "*   GOSUB lines for you, and insert REMARK pointers
    at those    *";
750 PRINT "*   locations so you know where to write each
    subroutine.      *";
760 PRINT "*                        --  HIT ANY KEY  --
          *";
770 PRINT L$;
780 IF INKEY$=""GOTO 780
790 CLS:PRINT L$;
800 PRINT "*       The program lines written include error
    traps and       *";
```


Fig. 9-2. Program listing for Autoprogrammer Instructions. (Continued from page 105.)

```
810 PRINT "*   other helpful features that you do not have to
    program     *";
820 PRINT "*   yourself.  Although Automatic Programmer will
    not write   *";
830 PRINT "*   a complete program, it will get the basics out
    of the      *";
840 PRINT "*   way fast, and allow you to use your creativity
    where it    *";
850 PRINT "*   counts the most.
          *";
860 PRINT "*
            *";
870 PRINT "*         3.) Automatic Programmer can also be used,
    to a      *";
880 PRINT "*   limited extent, to proofread the programs you
    have writ-  *";
890 PRINT "*   ten.  It will check for misspelled keywords,
    mismatched  *";
900 PRINT "*   parentheses, and some other errors.
          *";
910 PRINT"**";TAB(63)"**";
920 PRINT "*                          --  HIT ANY KEY  --
          *";
930 PRINT L$;
```

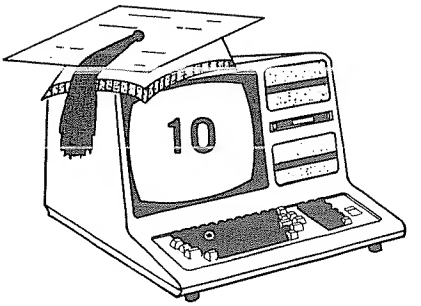
```
940 IF INKEY$="" GOTO 940
950 CLS:PRINT "**************************************************
    ****************";
960 PRINT "*      Please note:
         *";
970 PRINT"*";TAB(63)"*";
980 PRINT "*     o Program to be proofed must be saved in ASCII
    form.     *";
990 PRINT "*     o Program to be checked must NOT be 'packed'.
    There     *";
1000 PRINT "*       should be a space between the line number
    and the      *";
1010 PRINT "*       first statement in a line.  The module will
   work       *";
1020 PRINT "*       best if the target program is first run
     through one    *";
1030 PRINT "*       of the commercially available
     packing/unpacking rou-   *";
1040 PRINT "*       tines, which will insert spaces.
          *";
1050 PRINT "*    o  A cross reference list of 'bad' words and
     variables    *";
1060 PRINT "*       used in the program is compiled.  If Newdos
     80 2.0 or  *";
```


Fig. 9-2. Program listing for Autoprogrammer Instructions. (Continued from page 107.)

```
1070 PRINT "*       later is used,this list will be sorted.
        *";
1080 PRINT "*                        --  HIT ANY KEY  --
        *";
1090 PRINT L$;
1100 IF INKEY$="" GOTO 1100
1110 CLS:PRINT
1120 PRINT TAB(11)"********************************************* "
1130 PRINT TAB(11)"*                                           * "
1140 PRINT TAB(11)"*     Hit 'R' to repeat instructions.       * "
1150 PRINT TAB(11)"*              == Press ===                 * "
1160 PRINT TAB(11)"*          1.) To run Screen Editor         * "
1170 PRINT TAB(11)"*          2.) To run DB Starter            * "
1180 PRINT TAB(11)"*          3.) To run Program Proofer       * "
1190 PRINT TAB(11)"*                                           * "
1200 PRINT TAB(11)"********************************************* "
1210 A$=INKEY$:IF A$="" GOTO 1210
1220 IF A$="R" OR A$="r"  GOTO 90
1230 IF A$="1" THEN RUN"SCREEN/BAS"
1240 IF A$="2" THEN RUN "DBSTART/BAS"
1250 IF A$="3" THEN RUN "PROOFER/BAS"
1260 GOTO 1210
```

Fig. 9-2. Program listing for Autoprogrammer Instructions. (Continued from page 109.)

Now, wasn't that easy? When the user specifies HELP in one of the Automatic Programmer modules, a branch to a line that reads RUN "AUTOPROG/DOC" will take place. This program will then be loaded, and display the introduction to the other programs. At the end, an INKEY$ loop will accept one of three menu choices, loading and RUNning one of the three Auto Programmer modules. That's all there is to it. After a look at Figs. 9-1 and 9-2, class is dismissed for recess.



Visual Maker

Though photographic in nature, conventional slide shows used in business presentations rely more on text material, charts, and graphs, than on actual pictorial subjects. "Visual Maker" is a pair of programs written for the TRS-80 Model I/III and 4 that allows designing a series of text and graphics "frames," specifying how long each should appear on the CRT screen, and assembling them into a finished slide show.

Absolutely no user programming is required. The operator simply "draws" on the CRT screen, using the arrow keys for cursor control, placing alphanumeric characters and two types of graphic blocks as desired. Then, when the <ENTER> key is hit, and that frame is stored to disk. When all desired frames are assembled, a second program is run. The operator is given the opportunity to load and briefly check frames, in order, and then specify how many seconds each should appear on the screen. Then, a BASIC program is written that will display the frames as desired in a completed, ready-to-run slide show.

Visual Maker is similar in concept to Screen Editor, which writes BASIC subroutines that reproduce desired instructional screens. In fact, I used Screen Editor to write all the instructions in Visual Maker. The idea is to allow the user to enter various parameters, and then have the computer generate BASIC code automatically.

Visual Maker is very flexible in the ways users may assemble

A$	Used in INKEY$ loop.
AN$	Used in INKEY$ loop.
B	Beginning of video memory.
B1	ASC value of A$.
C	Cursor character.
C4	Line number being PEEKed.
E	End of video memory.
EFLAG	End of character line.
F$	Name of output file.
FLAG	Autonumbering flag.
FR	Frame counter.
'N	Loop counter.
N1-N3	Loop counters.
PR$(n)	Program lines stored in this array.
SP	Space.
T	Value found by current PEEK.
T2	Position of cursor.
Z	POKE position for cursor.

Fig. 10-1. Variables used in Visual Maker.

slide shows. Using the first program of the pair, a broad "library" of frames may be created that can be linked together by the assembler program in any order desired. This is very similar to the way in which many photographic slide shows are put together. Corporate communications departments draw heavily on an existing stock file of slides, minimizing the number of new frames that must be created for a given show. The same basic material can be used to develop a program tailored for employees, stockholders, directors, and the public.

AN$	Used in INKEY$ loop.
F$	Filename of output file.
FR	Frame counter.
IC	Increment to increase line number.
LN	Line number.
N$	Name of next slide to be added to show.
N2-N3	Loop counters.
PR$	Program line being built.
SC$	Seconds to display frame.
SP$	Space.
TM$(n)	Stores show program lines.

Fig. 10-2. Variables used in Show Assembler.

```
10 ' ********************************
20 ' *                              *
30 ' *       Visual Maker           *
40 ' *                              *
50 ' ********************************
60 CLEAR 10000
70 DEFINT A-Y
80 DIM PR$(16)
90 CLS
100 PRINT
110 PRINT TAB(9)"*************************************************************** "
120 PRINT TAB(9)"*    This program may be used to write subrou-  * "
130 PRINT TAB(9)"*    tines that will reproduce instructional    * "
140 PRINT TAB(9)"*    screens such as this one.                  * "
150 PRINT TAB(9)"*    You may build new screens from scratch,    * "
160 PRINT TAB(9)"*    edit existing ones, and then assemble      * "
170 PRINT TAB(9)"*    the screens or 'frames' into a slide       * "
180 PRINT TAB(9)"*    show.  The assembler module will ask you   * "
190 PRINT TAB(9)"*    how long you want each 'slide' to appear   * "
200 PRINT TAB(9)"*    on the screen.  The finished show will     * "
210 PRINT TAB(9)"*    be ready to run.                           * "
220 PRINT TAB(9)"*                                               * "
230 PRINT TAB(9)"*          HIT ANY KEY TO CONTINUE :            * "
```

```
240 PRINT TAB(9)"************************************************ "
250 A$=INKEY$:IF A$="" GOTO 250
260 CLS
270 PRINT
"*************************************************************";
280 PRINT "*                              Commands:
   *";
290 PRINT "*    Right Arrow          :  Move cursor one space right.
   *";
300 PRINT "*    Shift Right Arrow    :  Move cursor to far right of
   row.*";
310 PRINT "*    Left Arrow           :  Move cursor one space left.
   *";
320 PRINT "*    Shift Left Arrow     :  Move cursor to far left of row.
   *";
330 PRINT "*    Up Arrow             :  Move cursor one row up.
   *";
340 PRINT "*    Shift Up Arrow       :  Move cursor to top of screen.
   *";
350 PRINT "*    Down Arrow           :  Move cursor one space down.
```

Fig. 10-3. Program listing for Visual Maker.

```
370 PRINT "*     @                    :  Print graphics block -
"CHR$(191)"           *";
380 PRINT "*     &                    :  Print graphics block -
";CHR$(149);"         *";
390 PRINT "*       < ENTER >          :  Process the frame.
   *";
400 PRINT "*                       -- Hit any key --
   *";
410 PRINT
"*************************************************************";
420 A$=INKEY$:IF A$="" GOTO 420
430 CLS:PRINT:PRINT:PRINT:PRINT
440 PRINT TAB(16)"********************************* "
450 PRINT TAB(16)"*     Do you want automatic    * "
460 PRINT TAB(16)"*     frame numbering?         * "
470 PRINT TAB(16)"*         (Y/N)                * "
480 PRINT TAB(16)"*      -- Enter Choice --      * "
490 PRINT TAB(16)"********************************* "
500 AN$=INKEY$:IF AN$="" GOTO 500
510 IF AN$="Y" OR AN$="y" FLAG=1:FR=1
520 CLS:PRINT:PRINT:PRINT:PRINT
530 PRINT TAB(16)"********************************* "
540 PRINT TAB(16)"*     Do you wish to:          * "
```

```
550 PRINT TAB(16)"*       (C)reate a frame.      * "
560 PRINT TAB(16)"*       (E)dit a frame.        * "
570 PRINT TAB(16)"*       (Q)uit session.        * "
580 PRINT TAB(16)"*                              * "
590 PRINT TAB(16)"*      -- Enter Choice --      * "
600 PRINT TAB(16)"********************************* "
610 AN$=INKEY$:IF AN$="" GOTO 610
620 IF AN$="Q" OR AN$="q" END
630 IF AN$="C" GOTO 680
640 IF AN$="c" GOTO 680
650 IF AN$="e" GOTO 680
660 IF AN$="E" GOTO 680
670 GOTO 610
680 CLS:PRINT:PRINT:PRINT:PRINT
690 PRINT TAB(19)"************************* "
700 PRINT TAB(19)"*    Enter a file name  * "
710 PRINT TAB(19)"*    for this frame     * "
720 IF FLAG=1 PRINT TAB(19)"*    series.              * "
730 PRINT TAB(19)"*                       * "
740 PRINT TAB(19)"************************* "
750 PRINT TAB(25)"";:LINEINPUT" Filename: ";F$
760 F$=LEFT$(F$,8)
```

Fig. 10-3. Program listing for Visual Maker. (Continued from page 115.)

```
770 IF FLAG=1 THEN F$=LEFT$(F$,5)+MID$(STR$(FR),2)
780 IF AN$="C" OR AN$="c" CLS:GOTO 890
790 OPEN "I",1,F$
800 CLS
810 FOR N=1 TO 15
820 INPUT #1,PR$(N)
830 PRINT PR$(N);
840 NEXT N
850 INPUT #1,PR$(16)
860 PR$(16)=LEFT$(PR$(16),LEN(PR$(16))-1)
870 PRINT PR$(16);
880 CLOSE
890 B=15360:E=16384
900 Z=B
910 C=43
920 POKE Z,C
930 SP=32
935 ' *** Check Keyboard for Arrow input or character ***
940 A$=INKEY$:IF A$="" GOTO 940
950 B1=ASC(A$)
960 IF B1=A GOTO 1090
970 IF A$=CHR$(13) GOTO 1400
980 IF A$=CHR$(91) GOTO 1120
```

```
990 IF A$=CHR$(25) GOTO 1600
1000 IF A$=CHR$(24) GOTO 1680
1010 IF A$=CHR$(26) GOTO 1760
1020 IF A$=CHR$(27) GOTO 1840
1030 IF A$=CHR$(10) GOTO 1190
1040 IF A$=CHR$(9) GOTO 1260
1050 IF A$=CHR$(8) GOTO 1330
1060 IF B1=64 THEN B1=191
1070 IF B1=38 THEN B1=149
1080 A=B1
1090 POKE Z,A
1100 IF Z+1<E THEN Z=Z+1:POKE Z,C
1110 GOTO 940
1115 ' *** Cursor Up ***
1120 IF Z-64<B GOTO 940
1130 POKE Z,SP
1140 Z=Z-64
1150 Z1=Z-15360:IF Z1/32=INT(Z1/32) THEN C=191
1160 POKE Z,C
1170 C=43
1180 GOTO 940
```


Fig. 10-3. Program listing for Visual Maker. (Continued from page 117.)

```
1185 ' *** Cursor Down ***
1190 IF Z+64=>E GOTO 940
1200 POKE Z,SP
1210 Z=Z+64
1220 Z1=Z-15360:IF Z1/32=INT(Z1/32) THEN C=191
1230 POKE Z,C
1240 C=43
1250 GOTO 940
1255 ' *** Cursor Right ***
1260 IF Z+1>E GOTO 940
1270 POKE Z,SP
1280 Z=Z+1
1290 Z1=Z-15360:IF Z1/32=INT(Z1/32) THEN C=191
1300 POKE Z,C
1310 C=43
1320 GOTO 940
1325 ' *** Cursor Left ***
1330 IF Z-1<1 GOTO 940
1340 POKE Z,SP
1350 Z=Z-1
1360 Z1=Z-15360:IF Z1/32=INT(Z1/32) THEN C=191
```

```
1370 POKE Z,C
1380 C=43
1390 GOTO 940
1395 ' *** Read Screen ****
1400 IF Z<>E-1 THEN POKE Z,SP:CU=1
1410 :  FOR N=0 TO 1023 STEP 64
1420 C4=C4+1
1430 :       FOR N1=N TO N+63
1440 :         N3=N3+1
1450 :         T=PEEK(N1+15360)
1460 :         POKE N1+15360,191
1470 :         PR$=PR$+CHR$(T):IF T<>32 THEN EFLAG=N3
1480 :       NEXT N1
1490 PR$(C4)=PR$:PR$=""
1500 :  NEXT N
1505 ' *** Save File to Disk ***
1510 OPEN "O",1,F$
1520 :  FOR N=1 TO 16
1530 :    PRINT #1,CHR$(34);PR$(N);CHR$(34);
1540 :  NEXT N
1550 CLOSE 1
```

Fig. 10-3. Program listing for Visual Maker. (Continued from page 119.)

```
1560 FR=FR+1
1570 GOTO 520

1575 ' *** Calculate Cursor Row ***

1580 T1=INT((Z-B)/64)+1
1590 RETURN

1595 ' *** Jump Cursor To Right Margin ***

1600 GOSUB 1580
1610 T2=(T1*64)+15359
1620 IF T2>E GOTO 940
1630 POKE Z,SP
1640 Z=T2
1650 POKE Z,C
1660 C=43
1670 GOTO 940

1675 ' *** Jump Cursor to Left Margin ***

1680 GOSUB 1580
1690 T2=(T1*64)+15296
1700 IF T2<B GOTO 940
1710 POKE Z,SP
1720 Z=T2
```

```
1730 POKE Z,C
1740 C=43
1750 GOTO 940
1755 ' *** Jump Cursor to Bottom of Screen ***
1760 GOSUB 1580
1770 T2=(16-T1)*64
1780 IF T2+Z>E GOTO 940
1790 POKE Z,SP
1800 Z=Z+T2
1810 POKE Z,C
1820 C=43
1830 GOTO 940
1835 ' *** Jump Cursor to Top of Screen ***
1840 GOSUB 1580
1850 T2=(T1-1)*64
1860 IF Z-T2<B GOTO 940
1870 POKE Z,SP
1880 Z=Z-T2
1890 POKE Z,C
1900 C=43
1910 GOTO 940
```


Fig. 10-3. Program listing for Visual Maker. (Continued from page 121.)

```
10 ' **********************
20 ' *                    *
30 ' *  Show Assembler    *
40 ' *                    *
50 ' **********************
60 CLEAR 10000
70 DIM TM$(16)
80 SP$=CHR$(32)
90 IC=10:LN=10:FR=1
100 GOTO 130

105 ' *** Increment Line Counter ***

110 LN=LN+IC:PR$=STR$(LN)+SP$
120 RETURN

125 ' *** Open File to be Assembled ***

130 CLS:PRINT:PRINT:PRINT:PRINT
140 PRINT TAB(14)"************************************ "
150 PRINT TAB(14)"*                                  * "
160 PRINT TAB(14)"*     Enter the name of the        * "
170 PRINT TAB(14)"*     slide show now being         * "
```

```
180 PRINT TAB(14)"*     assembled.                    * "
190 PRINT TAB(14)"*                                   * "
200 PRINT TAB(14)"************************************ "
210 PRINT:PRINT:PRINT TAB(16)"===> Enter name: ";
220 LINEINPUT F$
230 F$=LEFT$(F$,8)
240 OPEN "O",2,F$

245 ' *** Input name of next frame ***

250 CLS
260 PRINT:PRINT:PRINT:PRINT
TAB(14)"************************************ "
270 PRINT TAB(14)"*                                   * "
280 PRINT TAB(14)"*     Enter the name of the         * "
290 PRINT TAB(14)"*     next frame to be added        * "
300 PRINT TAB(14)"*     to the slide show.            * "
310 PRINT TAB(14)"*                                   * "
320 PRINT TAB(14)"*      ('Q' to quit.)               * "
330 PRINT TAB(14)"************************************ "
340 PRINT:PRINT
350 PRINT TAB(16)"===> Enter name: ";
```

Fig. 10-4. Program listing for Show Assembler.

```
360 LINEINPUT N$
370 IF N$="Q" OR N$="q" GOTO 1030

375 ' *** Enter How Long to Display ***

380 CLS
390 PRINT:PRINT:PRINT TAB(14)"************************************ "
400 PRINT TAB(14)"*                                  * "
410 PRINT TAB(14)"*     How long would you like      * "
420 PRINT TAB(14)"*     this frame to be displayed?  * "
430 PRINT TAB(14)"*                                  * "
440 PRINT TAB(14)"*                                  * "
450 PRINT TAB(14)"************************************ "
460 PRINT:PRINT
470 PRINT TAB(16)"===> Enter seconds: ";
480 LINEINPUT SC$
490 CLS

495 ' ****** Load Frame from Disk ******

500 OPEN "I",1,N$
510 :  FOR N=1 TO 16
```

```
520 :     INPUT #1,TM$(N)
530 :     PRINT TM$(N);
540 :     FOR N2=1 TO 50:NEXT N2
550 :   NEXT N
560 FOR N3=1 TO 100:NEXT N3
570 CLOSE 1
580 CLS
590 PRINT:PRINT:PRINT:PRINT
TAB(14)"************************************ "
600 PRINT TAB(14)"*                                  * "
610 PRINT TAB(14)"*      That frame okay?            * "
620 PRINT TAB(14)"*                                  * "
630 PRINT TAB(14)"*           (Y/N)                  * "
640 PRINT TAB(14)"*                                  * "
650 PRINT TAB(14)"************************************ "
660 PRINT:PRINT
670 PRINT TAB(16)"===> Enter      : ";
680 AN$=INKEY$:IF AN$="" GOTO 680
690 IF AN$="N" OR AN$="n" GOTO 250
700 IF AN$="Y" OR AN$="y" GOTO 720
710 GOTO 680
720 FR=FR+1
```

Fig. 10-4. Program listing for Show Assembler. (Continued from page 125.)

```
730 GOSUB 110
740 PR$=PR$+"CLS"
750 GOSUB 980

755 ' ****** Print to Disk ******

760 TM$(16)=LEFT$(TM$(16),63)
770 :  FOR N=1 TO 16
780 :    GOSUB 110
790 :    PR$=PR$+"PRINT "+CHR$(34)+TM$(N)+CHR$(34)+";"
800 :    PRINT PR$
810 :    PRINT #2,PR$
820 :    PR$=""
830 :  NEXT N
840 GOSUB 110
850 PR$=PR$+"T$=TIME$"
860 GOSUB 980
870 PR$=PR$+"SE$=STR$("+SC$+")"
880 GOSUB 980
890 PR$=PR$+"ST=VAL(RIGHT$(T$,2))"
900 GOSUB 980
910 PR$=PR$+"FT=ST+VAL(SE$)"
```

```
920 GOSUB 980
930 PR$=PR$+"IF FT>59 THEN FT=FT-60"
940 GOSUB 980
950 PR$=PR$+"IF VAL(RIGHT$(TIME$,2))<>FT GOTO "+STR$(LN)
960 GOSUB 980
970 GOTO 250
980 PRINT #2,PR$
990 PRINT PR$
1000 GOSUB 110
1010 RETURN
1020 GOSUB 110
1030 PR$=PR$+"CLS"
1040 PRINT #2,PR$
1050 PR$=""
1060 GOSUB 110
1070 PR$=PR$+"GOTO "+STR$(LN)
1080 PRINT #2,PR$
1090 CLOSE
```

Fig. 10-4. Program listing for Show Assembler. (Continued from page 127.)

```
20 CLS
40 PRINT"                                                                  ";
50 PRINT"                                                                  ";
60 PRINT"                                                                  ";
70 PRINT"                    This is the first in a series                 ";
80 PRINT"                    of frames being created to dem-               ";
90 PRINT"                    onstrate the capabilites of                   ";
100 PRINT"                    Visual Maker.                                ";
110 PRINT"                                                                 ";
120 PRINT"                                                                 ";
130 PRINT"                                                                 ";
140 PRINT"                                                                 ";
150 PRINT"                                                                 ";
160 PRINT"                                                                 ";
170 PRINT"                                                                 ";
180 PRINT"                                                                 ";
190 PRINT"                                                                 ";
200 T$=TIME$
210 SE$=STR$(10)
220 ST=VAL(RIGHT$(T$,2))
230 FT=ST+VAL(SE$)
240 IF FT>59 THEN FT=FT-60
250 IF VAL(RIGHT$(TIME$,2))<>FT GOTO  250
```

```
270 CLS
290 PRINT"
300 PRINT"
310 PRINT"                    Simply create frames using
320 PRINT"                    the screen editor, then
330 PRINT"                    assemble them into a finished
340 PRINT"                    show.
350 PRINT"
360 PRINT"
370 PRINT"
380 PRINT"
390 PRINT"
400 PRINT"
410 PRINT"
420 PRINT"
430 PRINT"
440 PRINT"
450 T$=TIME$
460 SE$=STR$(20)
470 ST=VAL(RIGHT$(T$,2))
480 FT=ST+VAL(SE$)
490 IF FT>59 THEN FT=FT-60
500 IF VAL(RIGHT$(TIME$,2))<>FT GOTO  500
510 CLS
```

Fig. 10-5. Example of program produced by Visual Maker.

Thus you may design several dozen or several hundred frames that can be used and reused in multiple slide programs. Existing frames can also be stored as a sort of visual "boilerplate" and edited to form entirely new slides—without creating the entire visual from scratch.

To use Visual Maker, the operator first is given the opportunity to review the commands the visual editor recognizes. The arrow keys move the cursor around the screen, with SHIFT plus arrow jumping the cursor to the far edges of the screen (top, left and right sides).

Striking an alphanumeric key reproduces that symbol on the screen, much like a word processing program. In addition, two different graphic blocks can be summoned by hitting the@key and the & key. An entirely new frame may be created, or the filename of an existing frame entered and that visual edited.

The screen editor written for Visual Maker is a fairly simple one. Exiting from a given screen line should only be done at a point in which a space already exists, otherwise the character in the cursor position will be erased, or the line can be finished. The cursor will wrap around to the next line. The graphic blocks can be used to build charts, graphs, and other material. When you are satisfied with the screen design, hit <ENTER>.

At this point, the program PEEKs each location of video memory and stores the 64 characters of each line in a string variable, PR$(n). See Fig. 10-1 and the listing (Fig. 10-3) in this chapter. The sixteen elements of PR$(n) correspond to the sixteen lines on the screen. The information about each frame is then stored on disk. The filename for the frame, F$, is assembled from a prefix supplied by the user and a frame number, FR, which is incrementd each time a frame is designed during a given session.

This program may be changed easily by the user. If you do not like the graphics blocks provided, change the definitions of them to any character you please. You may also redefine any other keys to any other characters or graphics blocks. Simply choose keys that you do not plan to use in your screens. Some examples are the !, ", #, and % keys. Then, add program lines. For example, to change the quotation mark to a graphics block, type in:

```
xxxx IF B1=34 THEN B1=148
```

The second program, Show Assembler, takes the frames you have developed and uses them to write a BASIC program that will

display the frames for the number of seconds you indicate. To use the program (Figs. 10-2 and 10-4) you must supply a name for the slide show being assembled and then enter the names of the frames you want in the proper order. The program also asks the number of seconds, generally from 2 to 30, that you want the frame displayed.

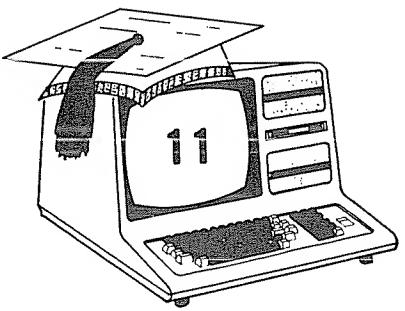
At this point the frame is loaded from disk and scrolled down your screen for a quick, last minute check. If it is indeed the frame you want, a complex series of procedures are carried out to write the necessary program lines to display the frame.

The lines assembled look something like this:

```
10  T=20 ' Time to display
20  T$=TIME$
30  SE$=STR$(T)
40  ST=VAL(RIGHT$(T$,2))
50  FT=ST+VAL(SE$)
60  IF FT>59 THEN FT=FT-60
70  IF VAL(RIGHT$(TIME$,2))< >FT GOTO 70
80  CLS
```

It is these lines which are assembled and printed to disk, substituting the actual time the user specified to display the frame for T. In this way, each successive frame will be displayed for the amount of time desired by the user. A sample program produced by Visual Maker is listed in Fig. 10-5.

Compared to DB Starter, Visual Maker is much more sophisticated, because it will write complete, ready-to-run slide show style programs, and several features have been added in screen editing over Screen Editor itself.



Global Replacer

Here is another program in the "REM-over" mold. This one, "Global Replacer," demonstrates how one program can be adapted to perform a second function. In concept the two are almost identical; instead of searching for remarks and then deleting them, however, the program looks for *any* string of the operator's choice. The string is then replaced with a second. The result is a global search-and-replace operation on a program, much like the same function in a word processing program.

Unlike some word processing programs, however, the user is shown each occurrence of the search string and offered the opportunity to replace it. You can pick and chose which to replace and which to leave alone. Variables and the listing are shown in Figs. 11-1 and 11-2, respectively.

The search string is input into S$ in line 90; since LINEINPUT is used, the string may contain commas and other string delimiters. The replacement string is entered into RE$. Then the input and output files are opened, and the first program (or text) line is loaded into A$ in line 260.

The user has been offered the option of deciding whether or not the program queries before making the replacement. A search routine basically identical to that used in REM-over hunts for the string. The difference is that line 290, where the former program had 'R=INSTR(P,A$,"REM")', Global Replacer substitutes S$ for

A$	Stores program line being searched.
B$, CH$	Used in INKEY$ loop.
E	Length of string being searched for.
F$	Filename of program being searched.
F1$	Name of output file.
L$	Left portion of program line.
N1	Loop counter.
P	Position to begin search.
R	Position of target string.
RE$	Replacement string.
S$	Target string.
Y$	String of spaces as long as replacement string.

Fig. 11-1. Variables used in Global Replacer.

REM. If R does not equal zero, then the line is cut into two sections. L$ stores everything in the line up to the beginning of the search string. R$ includes the rest of the line *after* the search string. Another string, Y$, is constructed from a series of blanks equal in length to the replacement string.

If the user has specified querying, control goes to line 360, where an INKEY$ loop awaits keyboard input. Each time through the loop L$, Y$, and R$ are printed on the same line, and are, after a short delay, followed by L$, RE$, and R$, The result is a flashing display; the left and right portions of the program line remain on the screen, while the potential replacement flashes on and off in its place. A "Replace it?" prompt asks for a decision. The program will only replace the string if a "Y" is entered; any other key will leave the program line as it was. Once the string has been replaced, the program branches back to look further. If the search string is not found, the program line is printed to the disk in line 490, and a new program line fetched.

Global Replacer is a short but powerful program that will let you make changes rapidly in a given program. Should you decide to change the name of a variable, substitute one keyword for another (LPRINT for PRINT, for example), or change some prompts and other material within quotes, it will handle them all. Its chief advantage over using a text editor for the same chore is the ability to examine each line before making the change. Those without word processing programs can use this utility, too.

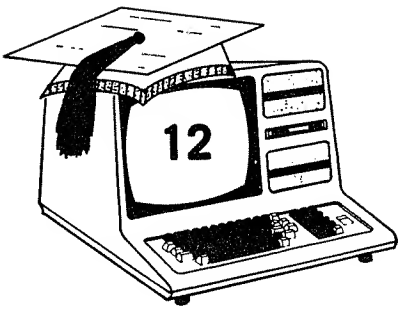
```
10 ' ****************
20 ' *                *
30 ' *    GLOBAL      *
40 ' *                *
50 ' ****************
60 CLEAR 5000
65 ' *** Set up Parameters ***
70 CLS:PRINT:PRINT
80 PRINTTAB(12)"Enter name of program to be processed :"
90 LINEINPUT F$
100 CLS:PRINT:PRINT
110 PRINTTAB(18)"Enter string to search for :"
120 LINEINPUT S$
130 CLS:PRINT:PRINT
140 PRINTTAB(17)"Enter string to replace with :"
150 LINEINPUT RE$
160 CLS:PRINT:PRINT
170 PRINTTAB(9)"Do you want to choose whether to replace each?"
180 PRINTTAB(29)"(Y/N)"
190 CH$=INKEY$:IF CH$="" GOTO 190
200 IF CH$="Y" OR CH$="y" THEN CH=1
210 CLS
```

```
220 Fl$=LEFT$(F$,8)+"/GBL"
225 ' *** Open Disk Files ***
230 OPEN "I",1,F$
240 OPEN "O",2,Fl$
250 IF EOF(1) GOTO 520
255 ' *** Load a line ***
260 LINEINPUT #1,A$
270 IF CH=1 THEN CLS
280 P=1
290 R=INSTR(P,A$,S$)
300 IF R=0 GOTO 490
310 L$=LEFT$(A$,R-1)
320 E=LEN(S$)
330 R$=MID$(A$,R+E)
340 Y$=STRING$(LEN(RE$),32)
350 IF CH=0 THEN GOTO 460
355 ' *** Replace it? ***
360 B$=INKEY$
370 PRINT @257,L$;Y$;R$
380 FOR Nl=1 TO 50:NEXT
```

Fig. 11-2. Program listing for Global Replacer.

```
390 PRINT @257,L$;RE$;R$
400 FOR N1=1 TO 50:NEXT
410 PRINT @ 788,"Replace it? (Y/N)"
420 IF B$="" GOTO 360
430 IF B$="Y" OR B$="y" GOTO 460
440 P=INSTR(P,A$,S$)+LEN(S$)-1
450 GOTO 290
460 A$=L$+RE$+R$
470 P=INSTR(P,A$,RE$)+LEN(RE$)-1
480 GOTO 290
485 ' *** Print to disk ***
490 PRINT#2,L$;RE$;:A$=R$
500 IF CH=0 THEN PRINT A$
510 GOTO 250
520 CLOSE
525 ' *** Do it again? ***
530 PRINT:PRINT
540 PRINTTAB(21)"Process another file?"
550 PRINTTAB(29)"(Y/N)"
560 A$=INKEY$:IF A$="" GOTO 560
570 IF A$="Y" OR A$="y" THEN RUN ELSE CLS
```

Fig. 11-2. Program listing for Global Replacer. (Continued from page 137.)



Menu Master

Strictly speaking, "Menu Master" is not a program-writing utility. It won't generate any lines for you, nor change an existing program. However, it may speed your initial work somewhat, and it can be interfaced with a variety of BASIC programs to save you time.

Menu Master is an all-purpose menu program that allows you to summon any of 26 (or more) programs, functions, or commands at the press of a single key. With it on your system disk, and an AUTO command to load BASIC and run Menu Master on powerup, it is possible to switch on your TRS-80 and go directly to your word processing program, format a disk, or perform some other task by hitting only one key.

As listed, the program does 26 things that I judged most useful to me, as shown in Fig. 12-1, but you can substitute those more suited to your own needs. Again, Menu Master takes advantage of some of the features of NEWDOS/80. You may want to make several changes to adapt the program to your own DOS.

The 26 menu choices are displayed in two columns of 13 each. Every choice is preceded by a single letter or number; the first 10 are invoked by pressing numerals from zero through nine, while the last 16 require pressing a letter from A to P. Either lowercase or uppercase letters are fine.

The menu choices are sometimes a logical, sometimes eclectic. Pressing zero through three summons the directories of Drives zero to three. Since the numbers correspond to the drives, this

series is easy to remember. If you have fewer than four drives, you can substitute a command of your choice.

Pressing other keys will command the system to copy a disk, change the name of a file, copy a single file from Drive :0 to Drive :1, or format a disk. You may also purge a disk, change a disk's name, or fiddle with the SYSTEM or DRIVE specifications.

If your computer is going to sit around all day, you can even turn on the clock and ask to be reminded when a certain time has passed. Should none of these please you, feel free to substitute commands of your own. The basic work has been accomplished for you. See the variable chart in Fig. 12-2, and the listing in Fig. 12-3.

On entering the program, the user should make some substitutions in lines 80-110 for his or her own word processing, communications, or spelling checker programs. The menu is then displayed (after a check) in line 160, to see if the timer has been set. If it has, the time for which the alarm is set is displayed at PRINT@ position 11, which is directly below the clock display on the Model I/III. Model 4 users can change this to suit.

An INKEY$ loop awaits input and, to help out, a flashing cursor is printed after the ENTER CHOICE prompt by a routine at lines 340-370. While waiting for the user to press a key, the program also repeatedly compares the current time, in line 380, with the time for which the alarm is set.

Once a key is pressed, the program looks to see that only one of the valid key choices has been entered. If a lowercase letter has been pressed (i.e., C>96), then it is converted to uppercase. The key depressed is used to send control to one of the subroutines that carry out the desired function.

```
0.) DIR :0                        D.) RUN Basic program
1.) DIR :1                        E.) Load Basic program
2.) DIR :2                        F.) Go to Basic
3.) DIR :3                        G.) Go to DOS
4.) Copy a disk                   H.) FREE
5.) Word Processing               I.) Purge disk
6.) Communications                J.) Format disk :1
7.) Run Spelling Checker          K.) Format disk :0
8.) Change name of file           L.) Change name of disk
9.) Copy file from :0 to :1       M.) Change SYSTEM
A.) Process a file for Comm       N.) Change DRIVES
B.) Turn computer clock off       O.) Turn clock on
C.) KILL a file from a disk       P.) Reboot System

              ENTER CHOICE :      =======>
```

Fig. 12-1. Sample screen from Menu Master.

A$	Used in INKEY$ loop.
AN$	Used in INKEY$ loop.
B$	Used with CMD to carry out function.
C	Value of user choice.
CG$	Changes to SYSTEM or PDRIVE.
CH	User choice.
CM$	Name of user text processor.
CO$	Name of user communications program.
D$	Drive number.
G$	Used with CMD to carry out function.
I	Position to POKE cursor.
N	Loop counter.
N1$	Name of file to be changed.
N2$	New name of the file.
PR$	Name of program.
R$	Used with CMD to carry out function.
SPELL$	Name of user spelling checker.
TN$	Time now.
TU$	Time up.
WP$	Name of user word processing program.

Fig. 12-2. Variables used in Menu Master.

The rest of the program consists of modules to do the task requested. In many cases the chore is a DOS function that, with NEWDOS/80, is achieved by assembling a string, such as R$, containing the command. For example, if the user presses zero, R$ will equal "DIR:0". A simple CMD R$ line will implement the DIR command. Other operating systems may use SYSTEM, or some variation to do this task. (Hint: use Global Replacer to make this change if you discover it after the program has been keyed in.)

The clock routine beginning at line 1000 required the most programming. The user is asked what time he or she wishes to be alerted, and the current time. The time is set, and TU$ given the value of the time for which the alarm is set.

You could probably squeeze in more than 13 menu choices. Model 4 computers, with 24 available screen lines should top out at 48 possibilities. Three columns of choices could up that to 72. You might be hard pressed to find enough unique keys, and could have to resort to upper and lowercase menu labels. But do you really think you could come up with 72 different things for your computer to do? Many of us don't even own that many programs.

```
10 ' ****************
20 ' *                *
30 ' * Menu Master *
40 ' *                *
50 ' ****************
60 CLEAR 1000
70 DEFINT A-Z

75 ' *** Parameters ***

80 WP$="Name of Your Word Processing System"
90 CO$="Name of your Communications Program"
100 SPELL$="Name of your Spelling Checker"
110 CM$="Name of your Text processor"
120 ON ERROR GOTO 1760
130 I=1003
140 C1=131

145 ' *** Display Menu ***

150 CLS:PRINT
160 IF TU$<>""PRINT @ 111,"* SET ";TU$;":00"
170 PRINT
```

```
180 PRINT "0.) DIR :0";TAB(30)"D.) RUN Basic program"
190 PRINT "1.) DIR :1";TAB(30)"E.) Load Basic program"
200 PRINT"2.) DIR :2";TAB(30)"F.) Go to Basic"
210 PRINT"3.) DIR :3";TAB(30);"G.) Go to DOS"
220  PRINT "4.) Copy a disk";TAB(30)"H.) FREE"
230 PRINT"5.) Word Processing";TAB(30)"I.) Purge disk"
240 PRINT"6.) Communications";TAB(30)"J.) Format disk :1"
250 PRINT"7.) Run Spelling Checker";TAB(30)"K.) Format disk :0"
260 PRINT"8.) Change name of file";TAB(30)"L.) Change name of
disk"
270 PRINT"9.) Copy file from :0 to :1";TAB(30)"M.) Change
SYSTEM"
280 PRINT"A.) Process a file for Comm";TAB(30)"N.) Change
DRIVES"
290 PRINT "B.) Turn computer clock off";TAB(30)"O.) Turn clock
on"
300 PRINT "C.) KILL a file from a disk";TAB(30)"P.) Reboot
System"
310 PRINT
320 PRINTTAB(12)" ENTER CHOICE :   ======>";
330 CH$=INKEY$
340 POKE 15360+I,C1
```

Fig. 12-3. Program listing for Menu Master.

```
350 FOR N=1 TO 20:NEXT N
360 POKE 15360+I,32
370 FOR N=1 TO 20:NEXT N

375 ' *** Check to see if time is up ***
380 IF MID$(TIME$,10,5)=TU$ GOTO 1080
390 IF CH$="" GOTO 330
400 C=ASC(CH$)
410 IF C>47 AND C<59 GOTO 420 ELSE GOTO 430
420 CH=C-47:GOTO 470
430 IF C<65 OR C>112 GOTO 330
440 IF C>80 AND C<97 GOTO 330
450 IF C>96 THEN CH=C-86:GOTO 470
460 CH=C-54
470 ON CH GOTO
1140,1140,1140,1140,480,680,690,700,710,830,930,940,1210,1360,133
0,1350,540,550,610,1380,1420,1660,1460,1560,950,920

475 ' *** Execute Commands ***

480 CLS:PRINT
490 PRINT"Make sure disk to be copied is in lower drive"
```

```
500 PRINT"Put blank disk to be copied to in upper drive"
510 PRINT:PRINT"Hit any key when ready"
520 A$=INKEY$:IF A$="" GOTO 520
530 CLS
540 CMD"S"
550 CLS:PRINT
560 CMD"FREE"
570 PRINT:PRINT
580 PRINTTAB(12)"== Hit any key to continue =="
590 A$=INKEY$:IF A$="" GOTO 590
600 RUN
610 CLS:PRINT:PRINT
620 PRINT"Which drive to purge (0-3)
630 INPUT D$
640 R$="PURGE :"+D$
650 CMD R$
660 RUN
670 CMD"s=copy :0 :1,,fmt
680 R$="S="+WP$:CMD R$
690 R$="S="+CO$:CMD R$
700 R$="S="+SPELL$:CMD R$
710 CLS:PRINT
```


Fig. 12-3. Program listing for Menu Master. (Continued from page 143.)

```
720 GOTO 790
730 PRINT"Do you want to check directory first?"
740 A$=INKEY$:IF A$="" GOTO 740
750 IF A$="Y" OR A$="y" GOTO 760 ELSE RETURN
760 PRINT"Which directory (0-3)?"
770 A$=INKEY$:IF A$="" GOTO 770
780 R$="dir :"+A$:CMD R$:RETURN
790 GOSUB 730
800 LINEINPUT"Enter name of file to be changed?";N1$
810 LINEINPUT"Enter new name :";N2$
820 B$="rename "+N1$+" "+N2$:CMD B$:RUN
830 CLS:PRINT
840 PRINT"Copy file from Drive 0 to Drive 1"
850 GOSUB 730
860 LINEINPUT"Enter name of file in Drive 0 to be copied to
Drive 1 ";N1$
870 LINEINPUT"Enter new name, or hit ENTER to keep old name
";N2$
880 IF N2$="" THEN N2$=N1$
890 R$="copy "+N1$+":0 to "+N2$+":1"
900 CMD R$
910 RUN
920 CMD"S=BOOT"
```

```
930 RUN CM$
940 CMD"CLOCK,N":GOTO 150
950 CMD"CLOCK"
960 CLS:PRINT
970 PRINT "DO YOU WANT TO SET AN ALARM?"
980 AN$=INKEY$:IF AN$="" GOTO 980
990 IF LEFT$(AN$,1)<>"Y" GOTO 150
1000 CLS
1010 LINEINPUT"WHAT TIME DO YOU WANT TO BE ALERTED?   ";TU$
1020 IF VAL(TU$)<10 THEN TU$="0"+TU$
1030 LINEINPUT"WHAT TIME IS IT NOW?  ";TN$
1040 IF VAL(TN$)<10 THEN TN$="0"+TN$
1050 G$="TIME,"+TN$+":00"
1060 CMD G$
1070 GOTO 150
1080 CLS
1090 PRINT CHR$(23)
1100 PRINT "NOW !!!"
1110 PRINT STRING$(191,64)
1120 GOTO 1080
1130 IF INKEY$="" GOTO 1130
1140 CLS:G$="DIR :"+CH$
```

Fig. 12-3. Program listing for Menu Master. (Continued from page 145.)

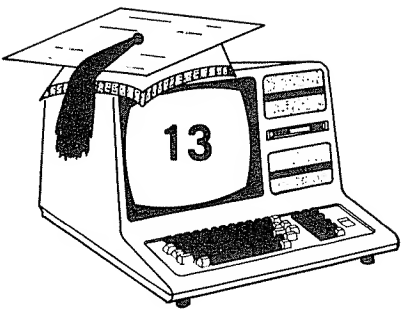
```
1150 CMD G$
1160 PRINT:PRINT
1170 PRINTTAB(15) "  -- HIT ANY KEY TO CONTINUE --"
1180 IF INKEY$="" GOTO 1180
1190 GOTO 150
1200 CLS:PRINT:PRINT
1210 CLS:PRINT
1220 GOSUB 730
1230 PRINT
1240 LINEINPUT"Enter name of program to be killed ";N1$
1250 PRINT"Which drive is it on (Hit ENTER for any)"
1260 A$=INKEY$:IF A$="" GOTO 1260
1270 IF A$=CHR$(13) THEN A$="":GOTO 1290
1280 A$=":"+A$
1290 R$="KILL "+N1$+A$
1310 CMD R$
1320 RUN
1330 CLS:PRINT:PRINT:LINEINPUT "ENTER NAME OF PROGRAM TO BE
LOADED :";PR$
1340 LOAD PR$
1350 NEW
1360 CLS:PRINT:PRINT:LINEINPUT "ENTER NAME OF PROGRAM TO BE RUN
:";PR$
```

```
1370 RUN PR$
1380 CLS:PRINT:PRINT
1390 R$="FORMAT :1"
1400 CMD R$
1410 RUN
1420 CLS
1430 R$="FORMAT :0"
1440 CMD R$
1450 RUN
1460 CLS:PRINT:PRINT
1470 PRINT"Which drive to change SYSTEM"
1480 INPUT D$
1490 R$="SYSTEM :"+D$
1500 CMD R$
1510 PRINT:PRINT"ENTER CHANGES :"
1520 LINEINPUT CG$
1530 R$=R$+CHR$(32)+CG$
1540 CMD R$
1550 RUN
1560 CLS:PRINT:PRINT
1570 PRINT"Which drive to change :"
```

Fig. 12-3. Program listing for Menu Master. (Continued from page 147.)

```
1580 INPUT D$
1590 R$="PDRIVE :"+D$
1600 CMD R$
1610 PRINT:PRINT"ENTER CHANGES :"
1620 LINEINPUT CG$
1630 R$=R$+CHR$(32)+CG$
1640 CMD R$
1650 RUN
1660 CLS:PRINT:PRINT
1670 PRINT"WHICH DISK TO CHANGE NAME"
1680 INPUT D$
1690 PRINT"ENTER NEW NAME :"
1700 LINEINPUT N$
1710 R$="PROT :"+D$+" NAME="+N$
1720 CMD R$
1730 RUN
1740 SAVE"MENU:0"
1750 GOTO 1130
1760 RESUME 150
```

Fig. 12-3. Program listing for Menu Master. (Continued from page 149.)



Lister

Lister combines some of the features of programs introduced previously. Like many of them, it loads a program and looks at each line. Then it examines the contents and performs some small trick that we programmers will find of value. In this case, it will format program listings into paged, neater groups.

The program asks the user to enter the name of the file to be listed on the line printer. The page width in columns is entered, along with the number of lines per page. Then, the file is opened and a line input into A$.

Then the program enters a FOR-NEXT loop that begins 10 characters to the left of the desired column width. That is, if 50 columns are desired, the program starts checking a line to be listed at the 40th character. This is considered the "hot" zone, where the program begins looking for either a colon or a space. When one is found, it splits the target program line at the colon or space and LPRINTS the two parts, with some spaces added to indent the second portion of the line past the line number above. The counter for the number of lines printed so far, LL, is also incremented. Whenever LL is greater than the desired number of lines per page, a new page is started, with an appropriate heading.

Note: because some computer setups hang up when attempts are made to LLIST without a printer being switched on or connected, leave the REMs in place while typing and debugging Lister. When everything is working fine, remove them and your listing will

```
10 ' **********************
20 ' *                    *
30 ' *    Word Counter    *
40 ' *                    *
50 ' **********************
60 CLEAR 4000
70 DEFINT A-Z
80 CLS:PRINT:PRINT
90 PRINT TAB(21)"Writer's Word Counter
   "
100 PRINT
110 PRINT TAB(6)"This program will
    count the number of actual words
    in a "
120 PRINT TAB(2)"text file, or any
    file that has been stored to disk
    in ASCII "
130 PRINT TAB(2)"format. In addition,
    it also provides the total number
    of "
140 PRINT TAB(2)"'standard ' five
    character words, and the average
    character "
150 PRINT TAB(2)"length of the words
    in the text. "
160 PRINT:PRINT TAB(17)"== Hit any
    key to continue == "
170 IF INKEY$="" GOTO 170
180 CLS:PRINT:PRINT' *** Access
    Disk File ***
```

Fig. 13-1. Example of listing producer by Lister.

A$	Stores program line being listed.
C$	Used in INKEY$ loop.
COL$	Width of printout.
L$	Name of file to be listed.
LL	Lines listed.
N	Loop counter.
P	Page number.
PG	Lines per page.
R$	Middle string of line being listed.

Fig. 13-2. Variables used in Lister.

go to the printer as well as to the screen. Figure 13-1 shows an example of another program in this book that has been LLISTed using Lister; variables and the listing are shown in Figs. 13-2 and 13-3.

```
10 ' ***********
20 ' *         *
30 ' *  Lister *
40 ' *         *
50 ' ***********
60 CLEAR 1000

65 ' *** Enter Parameters ***

70 CLS:PRINT:PRINT
80 PRINTTAB(16)"Enter name of file to be listed:"
90 LINEINPUT L$
100 PRINTTAB(21)"How many columns wide?"
110 INPUT COL$
120 COL=VAL(COL$)
130 PRINTTAB(20)"How many lines per page?"
140 INPUT PG$
150 PG=VAL(PG$)
160 P=1
170 GOSUB 410

175 ' *** Open Disk File ***
```

```
180 OPEN "I",1,L$
190 IF EOF(1) GOTO 350
200 IF LL>PG GOSUB 410

205 ' *** Look For Space or Colon ***

210 LINEINPUT#1,A$
220 :  FOR N=COL-10 TO COL
230 :    R$=MID$(A$,N,1)
240 :    IF R$=CHR$(32) GOTO 290
250 :    IF R$=":" GOTO 290
260 :  NEXT N
270 PRINT A$:'lprint a$
280 GOTO 190
290 L$=LEFT$(A$,N)
300 PRINT L$:'lprint l$:ll=ll+1
310 PRINT STRING$(5,32);:'lprint string$(5,32);
320 A$=MID$(A$,N+1)
330 IF A$="" GOTO190
340 GOTO 220
350 CLOSE
```


Fig. 13-3. Listing of Lister.

```
355 ' *** Do it again ? ***

360 PRINT:PRINT
370 PRINTTAB(23)"List another file?"
380 PRINTTAB(29)"(Y/N)"
390 A$=INKEY$:IF A$="" GOTO 390
400 IF A$="Y" OR A$="y" THEN RUN ELSE END

405 ' *** Page Routine ***

410 'LPRINT:LPRINT:LPRINT:LPRINT
420 PRINT:PRINT:PRINT"Please insert another page."
430 C$=INKEY$:IF C$="" GOTO 430
440 'lprint l$;" Listing Page ";P
450 LL=0
460 P=P+1
470 RETURN
```

Fig. 13-3. Listing of Lister. (Continued from page 155.)

Error Trapper

Error Trapper is dedicated to all of you who have written programs containing a bug or two. Our handy BASIC interpreters are nice enough to point them out to us at runtime. It would have been handy to have the syntax errors (at least) brought to our attention when the program line was first entered. But no, the computer is not that accommodating. It reserves judgment until we actually try to run the program.

Most amateur programs—and darned few professional BASIC programs—take advantage of the error trapping possibilities of the TRS-80. The machine not only tells us that there is an error, but in many cases it will point out exactly what type of error has been made. A clever code number is supplied, which can be manipulated by the program. In many cases some routine could be written to recover from the error. Or, in other cases, the error number could be used to supply the user-operator with some hint of what he or she has done wrong.

For example, a friendly prompt would be nice, something on the order of, "Program tried to divide by zero. Are you sure all the amounts you entered are correct?" Admittedly, many programmers don't understand enough about errors to do anything about them.

That's where Error Trapper comes in. Unlike Level II BASIC, Disk BASIC provides nice long error messages. Instead of "NF Error" we get NEXT WITHOUT FOR, which is much clearer. However, some of the more esoteric error messages may puzzle

```
10000 '      ****************************
10010 '      *                          *
10020 '      *      Error Trapper       *
10030 '      *                          *
10040 '      ****************************
10050 '
10060 CLS:PRINT
10070 ON ERROR GOTO 10080
10080 EC=ERR/2+1
10090 CLS
10100 IF EC>23 THEN EC=EC-27
10110 ON EC GOSUB
10120,10180,10220,10280,10340,10440,10520,10590,10680,10780,10870
,10940,10970,11030,11110,11180,11240,11290,11320,11360,11380,1144
0,11480,11530,11560,11590,11660,11700,11360,11360,11770,11360,113
60,11360,11800,11830,11910,11940,11360,12020,120
10120 PRINT "Next without For"
10130 PRINT:PRINT"Program got to NEXT without encountering FOR
first"
10140 PRINT"Check for incorrect GOTO.  Also, did you type GOTO
from"
10150 PRINT"COMMAND mode?"
10160 PRINT
```

```
10170 GOTO 12190
10180 PRINT"Syntax error"
10190 PRINT "Check for misspelled keywords, missing parentheses
or quotes"
10200 PRINT "as well as bad punctuation."
10210 GOTO 12190
10220 PRINT "Return Without Gosub"
10230 PRINT "Program may have gotten to a subroutine
improperly."
10240 PRINT "Check program lines immediately prior to this
subroutine"
10250 PRINT "to make sure program control does not allow
running"
10260 PRINT "into the following module."
10270 GOTO 12190
10280 PRINT "Out of Data"
10290 PRINT "Program was asked to Read more data items than
were"
10300 PRINT "available.  Check Data lines to be sure that none"
10310 PRINT "were left out by mistake.  FOR-NEXT loop may also"
10320 PRINT "be too large for number of items in Data."
10330 GOTO 12190
```


Fig. 14-1. Program listing for Error Trapper.

```
10340 PRINT "Illegal Function Call"
10350 PRINT "Program tried to perform an operation using an
illegal"
10360 PRINT "parameter.  Print the values of the variables in
the"
10370 PRINT "program line.  One will probably be a value that is
"
10380 PRINT "unsuited for one of the functions of that line."
10390 PRINT "For example, you might have PEEK(N) in the line,
and"
10400 PRINT "discover than N equals 70,000.   Or, in the case
of"
10410 PRINT "PRINT CHR$(N) that, through some error in the
program,"
10420 PRINT "N equals 256, or a larger number."
10430 GOTO 12190
10440 PRINT "Overflow"
10450 PRINT "A number is too large. If a variable is an
integer,"
10460 PRINT "this will occur if the number is larger than 32767"
10470 PRINT "single or double precision numbers can only be in
the range"
10480 PRINT "of about 1.7E+ (or minus) 38.  By changing a
```

```
variable "
10490 PRINT "from integer to single or double precision, most "
10500 PRINT "overflow errors will be avoided."
10510 GOTO 12190
10520 PRINT "Out of Memory"
10530 PRINT "Most likely, your program uses up too much memory "
10540 PRINT "because of very large arrays.  Cut down on array
size"
10550 PRINT "if possible.  Improperly nested branching routines"
10560 PRINT "(10 GOSUB 10, in the worst possible case) can also"
10570 PRINT "cause this, but rarely."
10580 GOTO 12190
10590 PRINT "Undefined line"
10600 PRINT "You typed a GOTO or GOSUB line, without entering "
10610 PRINT"the line where control was directed.  Or, in
editing,"
10620 PRINT "you killed a program section without the
corresponding"
10630 PRINT "line which called that section.   It is a good
idea"
10640 PRINT "to use a cross-reference utility to find out if a"
10650 PRINT" program line is called from elsewhere in a program"
```


Fig. 14-1. Program listing for Error Trapper. (Continued from page 159.)

```
10660 PRINT "before killing it."
10670 GOTO 12190
10680 PRINT "Subscript Out of Range"
10690 PRINT "Program tried to use an array element larger than
was"
10700 PRINT "DIMensioned.  Print out current value of the
subscript"
10710 PRINT "in the affected program line.  If it is 11, you
may"
10720 PRINT "have forgotten to DIMension that array, or you
have"
10730 PRINT "spelled the array name differently in the program
line."
10740 PRINT "For example:"
10750 PRINT "10 DIM ST$(20)"
10760 PRINT "20 S2$(12)=A$"
10770 GOTO 12190
10780 PRINT "Redimensioned Array"
10790 PRINT "Place DIM statements at beginning of program,
where"
10800 PRINT "they are not likely to be encountered more than
once."
10810 PRINT "If a program will be repeated, use the RUN command"
```

```
10820 PRINT"or make sure the GOTO directs control AFTER the DIM
statement."
10830 PRINT "If an array is being DIMensioned with a variable,"
10840 PRINT "(as in DIM A$(N)), make sure that the variable has
been"
10850 PRINT "assigned a value earlier in the program"
10860 GOTO 12190
10870 PRINT"Division by Zero"
10880 PRINT "Program error has produced a zero value in a
variable"
10890 PRINT"that is used in a division operation.  Check
variable"
10900 PRINT"to make sure it is not spelled incorrectly or that
the"
10910 PRINT"wrong variable is not being used.  Find  out why it
is"
10920 PRINT"zero when a value was expected."
10930 GOTO 12190
10940 PRINT"Illegal direct"
10950 PRINT "The INPUT command cannot be used as a direct
command."
10960 GOTO 12190
```


Fig. 14-1. Program listing for Error Trapper. (Continued from page 161.)

```
10970 PRINT"Type Mismatch"
10980 PRINT "Program tried to assign a string value to a
numeric"
10990 PRINT "variable or vice versa.  For example:  A$=A, or
A=CHR$(N)."
11000 PRINT "In most cases, these are caused by forgetting to
include"
11010 PRINT "the $ in a string variable or array."
11020 GOTO 12190
11030 PRINT "Out of String Space"
11040 PRINT "Larger CLEAR statement needed.  If none written,"
11050 PRINT "the system allocates only 50 bytes.  You may need
to"
11060 PRINT "to add a line reading CLEAR 500, or more,
depending"
11070 PRINT "on how much string space is consumed."
11080 PRINT"Model 4 mode dynamically allocates string space --"
11090 PRINT" you will not encounter this error message."
11100 GOTO 12190
11110 PRINT"String Too Long"
11120 PRINT"String variables and array elements can only be 255
bytes"
```

```
11130 PRINT "long.  Take string variables in program line, and "
11140 PRINT "find length by typing PRINT LEN(variable$) in
command"
11150 PRINT "mode.  The find why attempt was made to make this"
11160 PRINT "string that long."
11170 GOTO 12190
11180 PRINT "String Formula Too Complex"
11190 PRINT "Avoid such complex formulae as:"
11200 PRINT " A$=(LEFT$(MID$(A$,INSTR(B$,C$),LEN(A$))-1,)
LEN(A$)
11210 PRINT "Break operations down into several components.  You
will"
11220 PRINT "Never get all the parentheses in the right places,
anyway."
11230 GOTO 12190
11240 PRINT "Can't Continue"
11250 PRINT "Either you typed CONT after program had ended, or "
11260 PRINT "a program line was edited (thus ending the
program)."
11270 PRINT "You must start the RUN over."
11280 GOTO 12190
11290 PRINT "No Resume"
```


Fig. 14-1. Program listing for Error Trapper. (Continued from page 163.)

```
11300 PRINT "Program ended during error trapping."
11310 GOTO 12190
11320 PRINT"Resume Without Error"
11330 PRINT "You forgot, deleted, or bypassed the necessary "
11340 PRINT "ON ERROR GOTO message.  Place early in program."
11350 GOTO 12190
11360 PRINT"Unprintable error"
11370 GOTO 12190
11380 PRINT"Missing Operand"
11390 PRINT "Program neglected to include one of the necessary
operands."
11400 PRINT "Examples:"
11410 PRINT "A$=LEFT$(A$)"
11420 PRINT "POKE 15360 "
11430 GOTO 12190
11440 PRINT"Bad file data"
11450 PRINT "Data input from outside source, such as"
11460 PRINT "tape was incorrect or in wrong sequence."
11470 GOTO 12190
11480 PRINT"Disk Basic Only"
11490 PRINT "If not using Disk Basic, check commands to see
where"
11500 PRINT "incorrect command was entered.  Common example:"
```

```
11510 PRINT "INPUT #1, where INPUT #-1 is meant."
11520 GOTO 12190
11530 PRINT"Field Overflow"
11540 PRINT "More than 255 byes were allocated to a
random-access buffer."
11550 GOTO 12190
11560 PRINT"Internal error"
11570 PRINT "Whoops.  Disk operating system goofed."
11580 GOTO 12190
11590 PRINT "Bad File Number"
11600 PRINT "File buffer number that has not been assigned with
an"
11610 PRINT "OPEN statement was used.  Example:"
11620 PRINT "10 OPEN ";CHR$(34);"O";CHR$(34);",1,F$"
11630 PRINT "20 PRINT #2,A$"
11640 PRINT "Note that PRINT #1 should have been used instead."
11650 GOTO 12190
11660 PRINT"File Not Found"
11670 PRINT "File by that name not on disks currently in
drive(s)."
11680 PRINT "Or, you spelled filename wrong."
11690 GOTO 12190
```


Fig. 14-1. Program listing for Error Trapper. (Continued from page 165.)

```
11700 PRINT"Bad File Mode"
11710 PRINT "You tried to write to a buffer that had been opened
for"
11720 PRINT "input, or vice versa. Example:"
11730 PRINT "10 OPEN ";CHR$(34);"O";CHR$(34);",1,F$"
11740 PRINT "20 PRINT #1,A$"
11750 PRINT "Change the O to I"
11760 GOTO 12190
11770 PRINT"Disk I/O error"
11780 PRINT "OOOPS! Another computer error."
11790 GOTO 12190
11800 PRINT"Disk Full"
11810 PRINT "Insert new disk, or kill files."
11820 GOTO 12190
11830 PRINT"Input Past End"
11840 PRINT "Program tried to load more data from disk than was"
11850 PRINT "Available.  Check for empty file, or FOR-NEXT loop
that"
11860 PRINT "is too large.  With sequential files that grow,
adding"
11870 PRINT " an IF EOF(file buffer) GOTO xxx statement can
check"
11880 PRINT "for the end of the file, and send control to the
```

```
next"
11890 PRINT "module in an orderly manner."
11900 GOTO 12190
11910 PRINT"Bad Record Number"
11920 PRINT "Record number in a PUT statement larger than 1,340"
11930 GOTO 12190
11940 PRINT"Bad Filename"
11950 PRINT "Filename not legal.  Must conform to all rules for
naming"
11960 PRINT "programs or files in Disk Basic.  That is, cannot
be"
11970 PRINT "longer than eight characters,nor extension longer"
11980 PRINT "longer than three.  If variable being used for file
name"
11990 PRINT "check to make sure illegal value not being
assigned."
12000 PRINT "Error traps can be made to check for name
legality."
12010 GOTO 12190
12020 PRINT"Direct Statement In File"
12030 PRINT "You cannot load, run, or merge a disk file that is
not"
```


Fig. 14-1. Program listing for Error Trapper. (Continued from page 167.)

```
12040 PRINT "a Basic program.   This occurs when attempting to
load"
12050 PRINT "a text file stored in ASCII form or, possibly an
actual"
12060 PRINT "program that has had a line number removed from
the"
12070 PRINT "beginning of the line."
12080 GOTO 12190
12090 PRINT"Too Many Files"
12100 PRINT "Operating system will only permit 48 files and
programs"
12110 PRINT"on a single diskette."
12120 GOTO 12190
12130 PRINT"Disk Write Protected"
12140 PRINT "Write protect notch is covered.  Or disk is in
upside down!"
12150 GOTO 12190
12160 PRINT"File Access Denied"
12170 PRINT "Used wrong password."
12180 GOTO 12190
12190 PRINT"This error occurred in line ";ERL
12200 RESUME 10070
```

Fig. 14-1. Program listing for Error Trapper. (Continued from page 169.)

the best of us. Do you really know what sort of mistake will trigger an ILLEGAL DIRECT message?

This program, when appended to your own program, will spell it out for you. It provides *really long* error messages which, instead of just telling you how you goofed, will suggest situations that might have produced the error and places to check for the bug.

For example, if you see OUT OF DATA you know that the computer would like more data items. Error Trapper suggests that perhaps several data items were left out by mistake, or that the FOR-NEXT loop which reads the data is too large. ILLEGAL FUNCTION CALL suggests that the programmer list the offending line, and print out from command mode some of the values of the variables. Perhaps, it says, a number larger than 32767 was PEEKed, or you attempted to PRINT CHR$(256).

Little understood is how the TRS-80 manages to do something about errors. The secret is in line 10070, which is an ON ERROR GOTO command that summons the computer's interrupt routine. Interrupts are different than normal statements. If a program line says IF INKEY$=" " GOTO, it will act on that *only* at the exact moment that the line is interpreted by BASIC. In order to make INKEY$ work, we have to loop back, over and over, until something happens.

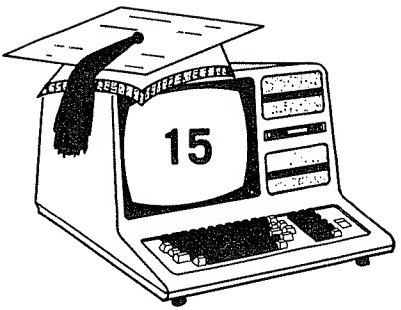
However, once ON ERROR has been activated, the computer can go on to other things. The program can perform all sorts of different functions, and the interrupt routine will remain dormant . . . until an error occurs. Then it will obey the command and send control to the line previously specified.

You can't even turn off the interrupt routine by exiting the program. Run Error Trapper and hit Break at some point. Then, trigger an error by typing in a syntax error or some other goof from command mode. Oops! The program is running again, and you are at line 10080. You didn't even type RUN. That is the interrupt routine at work.

Once an error has taken place, Error Trapper looks to see what kind of error it is. An error deposits a value in the reserved variable ERR. Dividing ERR by two and then adding one, we come up with a number that relates to the error code; the same error always produces the same unique number. We use that number in this program in an ON...GOSUB line that directs control to the appropriate error message. In a real program you might substitute for the message, some type of error trap or perhaps a routine that corrects the error.

For example, if the error were FILE NOT FOUND, you might write a routine that asks the user to check the filename, or deposit the correct disk in the drive. Then it would ask again for the filename. Using RESUME followed by a line number, control can be returned to the main body of the program.

If you append Error Trapper to your own programs, you will want to move the ON ERROR line higher in the program, so it will be activated BEFORE the main body of the program is run. Error Trapper is listed in Fig. 14-1.



Chain Zapper

Chain Zapper is aimed squarely at NEWDOS/80 users, but the idea behind it can also be adapted by users of other operating systems. To find out whether or not you qualify, ask yourself the following question: Do you flinch every time you open an envelope containing those familiar blue pages full of mandatory ZAPs for your favorite disk operating system? The ones that inform you that you must type in the following 47 bytes, plus half of SYS6/SYS, in order to avoid a horrible problem that may crop up if more than seventeen directory entries begin with the letter Q?

Those who faithfully apply ZAPs to their NEWDOS/80 operating system, or to programs in order to make them compatible with DOS, can now rely on their computer to take a good 50 percent of the drudgery out of this patching. Chain Zapper is a program which will create a custom chain file to do this chore.

Why not simply type in the ZAPs by hand, if they still must (obviously) be entered into some other utility program. There are a number of very good reasons. First, using Chain Zapper, you have only to type the filename to be zapped, the relative sector, first byte, and the actual ZAPs. This procedure can be repeated for any number of desired ZAPs in one session. More importantly, you can LOAD this chain file at any time, and proofread the patches you have typed in *before* the dirty work is done on the disk.

The real time savings come in applying the patches. The chain file created by Chain Zapper is activated merely by the normal

CHAIN command. If you have called up your ZAP file, NEWZAP, type CHAIN NEWZAP,ZAP (or DO NEWZAP,ZAP if using NEWDOS/80 2.0). In this case, ZAP is the section ID automatically tagged onto the file by Chain Zapper.

The chain file will load SUPERZAP, invoke DFS (display file's sectors), enter the correct sector—plus MOD, the starting byte, all the ZAPS, <ENTER> to finish the patching, and answer "Y" to the "Okay to write modification to disk?" prompt.

Then the chain file will return control to the main SUPERZAP menu and, if additional ZAPs have been included in the file, go on and patch the next, and the next. If all the disks containing the affected files are already loaded in the correct drives, the process can be amazingly fast. The CRT screen flashes almost quicker than the eye can follow.

In fact, the patching is so quick that it is a viable replacement for copying sectors to update all your disks. Once you have ascertained that a ZAP is correct (by proofreading before running the chain file), and have run one version a few times to make sure that everything seems to be okay, simply put other disks in drives and ZAP them all automatically. You may want to save an unaltered version, just in case. This is a technical term called "good data processing practice," or in layman's language, "covering your backside."

While Chain Zapper will save an individual much time, it has even more extensive application among user groups and computer clubs. One person can create the file and share it with all other members who are using NEWDOS/80. Copying disk sectors which have been ZAPped is time-consuming, because different file relative sectors may be in different locations on disks. But the Chain Zapper file doesn't care where the program to be patched starts on the disk. It invokes "DFS" for each file and relies on SUPERZAP to find the correct location. Once one member has written the chain file, it may be freely distributed for all to use.

This marvel of automation also makes it possible to goof on a truly mammoth scale. Be careful when entering ZAPs.

There are several modifications you may wish to make. By substituting OPEN "E" for OPEN "O" in line 180, and replacing the default "ZAP" section ID with a string variable that can be input with an ID name by the user, one chain file can serve a continuing series of ZAPS. When you receive your new patches, they can be added onto the end of the existing ZAP chain file. OPEN "E" opens a sequential file without resetting the EOF marker to zero, so the

new information is tacked onto the end. To invoke a specific set of patches, you'll have to use the appropriate section ID in place of the ZAP ID supplied with the unaltered program.

The program described in Figs. 15-1 and 15-2, makes use of no special routines. Some of the chain commands, such as SUPERZAP, DFS, and MOD, are built in, and are automatically written to the chain file with no user intervention. Other items, such as filespec, first byte to modify, and the actual ZAPs, must be supplied by the user. Apparat already formats all patches in filename, file relative sector, relative byte, new bytes order.

Patches may be entered as one long string, up to 255 characters in length. One space, and one only, must be entered between bytes. If you forget to space at the end of the patch, one will be added. The program looks for these spaces, using INSTR, to divide the ZAP into separate bytes. The bytes themselves are further parsed into nybbles and written to the chain file. Individual patches longer than 255 characters should be treated as two (or more) separate ZAPs. Following each patch, the program will ask if the user wishes to do another. If so, the input and disk write routines are repeated (except for the initial lines which invoke SUPERZAP).

The EDIT mode has purposely been kept fairly primitive, in order to discourage extensive use. You should be very careful when entering the original ZAPS. They are all displayed on the screen, so check them out before you hit ENTER. If a change must be made, however, the user may specify the chain zapping file to be edited, and type in the name of the affected program or other file. Chain Zapper will then search through the chain file, and stop at the *first*

A$	Used in INKEY$ loop.
E	Counter for name of file being edited.
F$(n)	File being edited.
F3$	Name of file to which ZAPs are being applied.
FI$	Name of output file.
G$	New value.
HEX$	Hex values.
I$	Used in INKEY$ loop.
N, N2	Loop counters.

Fig. 15-1. Variables used in Chain Zapper.

```
10 ' ****************
20 ' *              *
30 ' * Chain Zapper *
40 ' *              *
50 ' ****************
60 CLEAR 10000
70 DIM F$(500)

75 ' *** Menu ***

80 CLS:PRINT:PRINT
90 PRINT "Would you like to :"
100 PRINTTAB(10)"1.) Create new ZAP file"
110 PRINTTAB(10)"2.) EDIT existing ZAP file"
120 PRINT
130 PRINTTAB(6)"-- ENTER CHOICE --"
140 A$=INKEY$:IF A$="" GOTO 140
150 A=VAL(A$):IF A<1 OR A>2 GOTO 140
160 ON A GOTO 170,540

165 ' ****** CREATE ZAP CHAIN FILE ******

170 CLS:PRINT:PRINT
180 LINEINPUT"Enter name of ZAP file. Extension
    /JCL will be added automatically :";FI$
```

```
190 IF RIGHT$(FI$,4)="/JCL" GOTO 210
200 FI$=FI$+"/JCL"
210 OPEN "O",1,FI$
220 PRINT #1,CHR$(128)+"ZAP"
230 PRINT #1,"SUPERZAP"
240 PRINT #1,"DFS"
250 LINEINPUT "Enter filespec to be ZAPPED
    (You may include drive) :";F$
260 PRINT #1,F$
270 INPUT "Enter file relative sector :";F$
280 F=VAL(F$):IF F<0 GOTO 270
290 PRINT #1,F$
300 PRINT #1,"M"+CHR$(131)
310 PRINT #1,"O"+CHR$(131)
320 PRINT #1,"D"+CHR$(131)
330 INPUT"Enter first byte to modify :";F$
340 IF LEN(F$)<>2 GOTO 330
350 PRINT #1,LEFT$(F$,1)+CHR$(131)
360 PRINT #1,RIGHT$(F$,1)+CHR$(131)
370 INPUT "Enter new hex values. Space between each byte";HEX$
380 IF RIGHT$(HEX$,1)<>CHR$(32) THEN HEX$=HEX$+CHR$(32)
390 :  FOR N=1 TO LEN(HEX$) STEP 3
```

Fig. 15-2. Program listing for Chain Zapper.

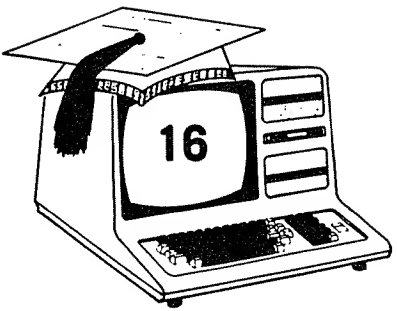
```
400 :     F$=MID$(HEX$,N,INSTR(HEX$,CHR$(32)))
410 :     PRINT #1,LEFT$(F$,1)+CHR$(131)
420 :     PRINT #1,MID$(F$,2,1)+CHR$(131)
430 :  NEXT N
440 PRINT #1,CHR$(13)+CHR$(131)
450 PRINT #1,"Y"+CHR$(131)
460 PRINT #1,CHR$(13)+CHR$(131)
470 PRINT #1,"X"+CHR$(131)
480 PRINT"Would you like to enter another ZAP?"
490 I$=INKEY$:IF I$="" GOTO 490
500 IF I$="Y" GOTO 240
510 PRINT #1,"EXIT"
520 CLOSE 1
530 GOTO 80
535 ****** EDIT ZAP CHAIN FILE ******
540 CLS:PRINT:PRINT
550 LINEINPUT"Enter name of file to be edited : ";FI$
560 OPEN "I",2,FI$
570 E=E+1
580 LINEINPUT #2,F$(E)
590 IF EOF(2) GOTO 610
600 GOTO 570
610 CLOSE 2
```

```
620 CLS:PRINT:PRINT
630 LINEINPUT"Enter name of file for which zaps to be edited
:";F3$
640 PRINT"Hit Space Bar to see next entry, 'C' to change"
650 :  FOR N=1 TO E
660 :    IF F$(N)=F3$ THEN FLAG=1
670 :    IF FLAG=0 GOTO 740
680 :    PRINT F$(N);" ";
690 :    I$=INKEY$:IF I$="" GOTO 690
700 :    IF I$<>"C" OR I$<>"c" GOTO 740
710 :    INPUT "Enter new value :";G$
720 :    IF RIGHT$(F$(N),1)=CHR$(131) THEN
         F$(N)=G$+CHR$(131):GOTO 740
730 :    F$(N)=G$
740 :  NEXT N
745 ' ****** WRITE EDITED FILE TO DISK ******
750 OPEN "O",1,FI$
760 :  FOR N2=1 TO E
770 :    PRINT #1,F$(N2)
780 :  NEXT N2
790 CLOSE 1
```

Fig. 15-2. Program listing for Chain Zapper. (Continued from page 177.)

patch for that particular program. Each byte will be displayed, and may be changed. If several separate ZAPs exist for a single program, you may have to page through several patches to find the one you want.

It is obvious, with postage and media costs as high as they are, that it would be impossible for a company like Apparat to send out chain files in disk or cassette form to do our patching for us. Their support in providing printed sheets for free (so far), is effort enough. However, given a wide enough distribution of chain files created by Chain Zapper, this tedious but necessary chore can be minimized for many more users.



Translator

Most of the BASIC language's limitations stem from its original purpose as a high-level language that would be easy for beginners to learn and use. Its strongest point—the simple English keywords—provides an artificial barrier for those whose primary language is not English. Some of the largest Spanish-speaking communities in the world, for example, are in the United States. The availability of a BASIC in Spanish might make it easier for these citizens to use computers at an earlier age.

A machine-language Spanish-BASIC interpreter for any of these would be ideal. Programs could be written in a Hispanic version of BASIC, run, tested, and debugged in that form. Unfortunately, that would be a major undertaking, best tackled by a software house with some hopes of recouping the time investment through sales—but one-tenth of a loaf is often better than none. Translator is a simple pseudo-compiler that converts programs written in Spanish tiny BASIC to standard BASIC for running.

In other words, the program is used to write the source code, using the Level I type of Spanish keywords, instead of the English BASIC equivalent. As each line is entered the program checks it for various criteria (must begin with a line number, with no more than one statement per line) and generates a new line of code, replacing each of the Spanish keywords with the English equivalent. Both versions may be saved to disk or listed at any time.

Translator combines some of the features of Global Replacer

and Program Proofer. It compares its internal list of allowable keywords with those in the input lines, and replaces them with the equivalents as needed.

Editing is accomplished by re-entering the line. The English ("compiled") version of the program is object code that may be loaded and run under your BASIC interpreter, like any BASIC program, as long as the code entered in Spanish conformed to the normal syntax rules of BASIC. Ideally, the program should be used by a person who already knows standard BASIC to teach a Spanish-speaking person how to program.

The Spanish words chosen are not necessarily the best possible equivalents for the BASIC keywords they replace. The BASIC translations were chosen using two criteria. The Spanish words had to be short and mean approximately what the BASIC equivalents mean. Because keywords have the effect of commands, the imperative form of the verbs was used. Second, programming was made easier by selecting Spanish words that were either the same length or longer than the BASIC keywords.

```
Spanish Version

10 IMPRIMA "PROGRAMMA"
20 ENTRE "SU NOMBRE :";A$
30 SI A$="DAVID" LUEGO IMPRIMA "HOLA
DAVID!"
40 SI A$<>"DAVID" VAYA SUB 100
50 FIN
100 IMPRIMA "HOLA,";A$
110 RETORNE

English Version

10 PRINT   "PROGRAMMA"
20 INPUT "SU NOMBRE :";A$
30 IF A$="DAVID" THEN  PRINT   "HOLA
DAVID!"
40 IF A$<>"DAVID" GOSUB    100
50 END
100 PRINT   "HOLA,";A$
110 RETURN
```

Fig. 16-1. Example of program produced by Translator.

A(n)	Difference in length of keywords.
A$	Line entered by user.
A1$	Used in INKEY$ loop.
B	Position of quote in line input.
C	Position of colon in line input.
COM$	Command entered by user.
CP$(n)	Array storing program lines in English.
CU	Counter.
E2$(n)	Array storing program lines in Spanish.
F$	Filename.
F3$	Filename.
FLAG	Shows whether instructions have been displayed.
G	Loop counter.
IG$	Program line input by user.
L	Length of program line.
N	Loop counter.
NE$	Name of program in Spanish.
NI$	Name of program in English.
P	Print @ position.
X	Set X coordinate.

Fig. 16-2. Variables used in Translator.

To use the program the student types RUN in English, and is shown a summary of the commands and statements available. This list can be summoned at any time by typing HELP or AYUDA at the " > " prompt. An existing program may be loaded from the disk using the CARGE command. Prompts ask for the name of the program in Spanish and English. Then, a program can be edited, or new lines added. Figure 16-1 shows both Spanish and English versions of a program written using Translator.

A specific line in Spanish can be seen at any time by entering ALISTE xxx, where xxx is the line number; by typing just ALISTE, the entire program will be presented, a section at a time. Entering LIST, in English, will display the compiled English version. NUEVO or CORRA will erase the current program in memory, and allow starting over.

Only line numbers between 1 and 200 may be used, and only single statements are allowed per line. Spaces must be used after line numbers and between words. It is permissible to end a line with a space, since one is added automatically. Spaces are essential, because in searching for keywords the program looks not for, say, the letters SI, but for <space>SI<space>. Otherwise, by the time

```
10 ' ***************
20 ' *             *
30 ' *  Translator *
40 ' *             *
50 ' ***************
60 CLEAR 1000
70 GOTO 210

75 ' *** Print Border ***

80 :   FOR N=1 TO 63
90 :    PRINT @ N, CHR$(159);
100 :  NEXT N
110 :  FOR N=833 TO 895
120 :    PRINT @ N, CHR$(190);
130 :  NEXT N
140 :  FOR X=0 TO 41
150 :    SET(1, X)
160 :    SET(127, X)
170 :  NEXT X
180 RETURN
190 PRINT @ P, CL$;
200 PRINT @ P,"";
```

```
205 ' *** Initialize ***

210 DEFINT A-Z
220 NW=21
230 L2=200
240 C1$=CHR$(34)
250 C2$=CHR$(58)
260 C3$=CHR$(32)
270 DIM A(21), E$(21), E2$(200), CP$(200), E3$(21), SP$(21)
280 CLS
290 RESTORE

295 ' *** Null arrays ***

300 :  FOR N=1 TO 200
310 :    E2$(N)=""
320 :    CP$(N)=""
330 :  NEXT N

335 ' *** Read Difference Data ***

340 :  FOR N=1 TO NW
```


Fig. 16-3. Program listing for Translator.

```
350 :    READ A(N)
360 :  NEXT N

365 ' *** Read Spanish and English keywords ***
370 :  FOR N=1 TO NW
380 :    READ E3$(N)
390 :    E3$(N)=C3$+E3$(N)+C3$
400 :    READ SPAN$(N)
410 :    SPAN$(N)=C3$+SPAN$(N)+C3$
420 :  NEXT N

425 ' *** Equalize length ***

430 :  FOR N=1 TO NW
440 :    E$(N)=E3$(N)+STRING$(A(N), 32)
450 :  NEXT N
460 DATA 0, 2, 0, 2, 0, 2, 1, 1, 1, 1, 0, 3, 2, 2, 1, 3, 1, 0,
1, 1, 1
470 DATA IF, SI, RUN, CORRA, INPUT, ENTRE, LIST, ALISTE, END,
FIN, PRINT, IMPRIMA, READ, LLEVE, DATA, DATOS, THEN, LUEGO, FOR,
PARA, STOP, CESE, NEXT, PROXIMO
480 DATA CLS, BORRE, GOTO, VAYA A, RESTORE, RESTAURE
490 DATA GOSUB, VAYA SUB, RETURN, RETORNE, ON, EN
```

```
500 DATA STEP, GRADA, REM, NOTA, LET, HACE
510 GOSUB 80
520 FLAG=1

525 ' *** Instructions ***

530 PRINT @ 144,"SPANISH-ENGLISH PROGRAM TRANSLATOR";
540 PRINT @ 264,"Do you want instructions (Y/N)?";
550 A1$=INKEY$
560 IF A1$="" THEN 550
570 IF A1$="Y" THEN 600
580 IF A1$="N" THEN CLS: GOTO 1430
590 GOTO 550
600 PRINT @ 264, STRING$(50, 32);
610 PRINT @ 264,"This program allows Spanish-speaking students
to ";
620 PRINT @ 328,"write programs using Spanish keywords instead
of";
630 PRINT @ 392,"the English equivalents.  Most LEVEL I-type
key-";
640 PRINT @ 456,"words may be used.";
650 PRINT @ 520,"The program prepares two versions of the
```


Fig. 16-3. Program listing for Translator. (Continued from page 185.)

```
program";
660 PRINT @ 584,"--  one in Spanish, and a 'translated ',
English";
670 PRINT @ 648,"version.";
680 PRINT @ 781,"   Hit any key to continue :";
690 IF INKEY$ ="" THEN 690
700 PRINT @ 136,"Although programs may be written in Spanish,
they";
710 PRINT @ 200,"may not be RUN in that form (this is not an
inter-";
720 PRINT @ 264,"preter) until they have been translated  into
Eng-";
730 PRINT @ 328,"lish-BASIC.
";
740 PRINT @ 392, STRING$(50, 32);
750 PRINT @ 456,"Both the Spanish and  English  versions may
be";
760 PRINT @ 520,"saved to disk under filenames of your choice.
The";
770 PRINT @ 584,"English version can then be loaded and RUN
norm-";
780 PRINT @ 648,"ally.     ";
790 IF INKEY$ ="" THEN 790
```

```
800 PRINT @ 136,"To use, type in program, using the Spanish
keywords";
810 PRINT @ 200,"where needed.  Only one statement is allowed
per   ";
820 PRINT @ 264,"line.  User MUST add a space after line numbers
  ";
830 PRINT @ 328,"and all words.  Only line numbers between 1 and
";
840 PRINT @ 392,"200 may be used.
";
850 PRINT @ 456, STRING$(50, 32);
860 PRINT @ 520,"To edit any line, just re-enter that line
number   ";
870 PRINT @ 584,"and the new line (like LEVEL I editing).
  ";
880 PRINT @ 648, STRING$(50, 32);
890 IF INKEY$ ="" THEN 890
900 PRINT @ 136,"Other disk BASIC keywords not translated may be
   ";
910 PRINT @ 200,"incorporated into the program if they adhere to
   ";
920 PRINT @ 264,"correct syntax.  These include :
 ";
```

Fig. 16-3. Program listing for Translator. (Continued from page 187.)

```
930 PRINT @ 328, STRING$(50, 32);
940 PRINT @ 392,"ELSE,INSTR,RIGHT$,LEFT$, as well as functions,
 ";
950 PRINT @ 456,"(INT,RND), operators (AND,OR).
   ";
960 PRINT @ 520, STRING$(50, 32);
970 PRINT @ 584,"If you have any questions type either 'HELP' or
   ";
980 PRINT @ 648,"'AYUDA'.  You will be shown a list like these:
 ";
990 IF INKEY$ ="" THEN 990
1000 PRINT @ 200, STRING$(50, 32);
1010 PRINT @ 584, STRING$(50, 32);
1020 PRINT @ 648, STRING$(50, 32);
1030 GOSUB 1220
1040 PRINT @ 136,"A typical program might look something like
this : ";
1050 PRINT @ 200, STRING$(50, 32);
1060 PRINT @ 264,"          10 ENTRE ";C1$;"COMO SE
LLAMA";C1$;";A$       ";
1070 PRINT @ 328,"          20 SI A$=";C1$;"JOSE";C1$;" VAYA A 40
       ";
1080 PRINT @ 392,"          30 CESE
```

```
      ";
1090 PRINT @ 456,"          40 IMPRIMA ";C1$;"HOLA JOSE";C1$;"
          ";
1100 PRINT @ 520,"          50 FIN
    ";
1110 PRINT @ 584, STRING$(50, 32);
1120 PRINT @ 648, STRING$(50, 32);
1130 PRINT @ 778,"Hit any key to run program
1140 IF INKEY$ ="" THEN 1140
1150 FLAG=0
1160 CLS
1170 GOTO 1430
1180 GOSUB 1200
1190 GOTO 1430
1200 CLS
1210 GOSUB 80
1220 PRINT @ 136," Los Mandados:
   ";
1230 PRINT @ 264,"Ahorre  (ahorrar una programma al disk)
   ";
1240 PRINT @ 328,"CARGE   (cargar una programma de disk)
   ";
```


Fig. 16-3. Program listing for Translator. (Continued from page 189.)

```
1250 PRINT @ 392,"ALISTE  (Alistar una programma en espanol)
  ";
1260 PRINT @ 456,"LIST    (Alistar una programma en ingles)
  ";
1270 PRINT @ 584,"AYUDA,CORRE,NUEVO,BORRE
  ";
1280 PRINT @ 714,"Empuje <ENTER>";
1290 IF INKEY$ ="" THEN 1290
1300 PRINT @ 136,"Las declaraciones:
  ";
1310 PRINT @ 264,"IF=SI          RUN=CORRA        INPUT=ENTRE
  ";
1320 PRINT @ 328,"END=FIN        LIST=ALISTE      PRINT=IMPRIMA
  ";
1330 PRINT @ 392,"READ=LLEVE     THEN=LUEGO       NEXT=PROXIMO
  ";
1340 PRINT @ 456,"DATA=DATOS     GOTO=VAYA A      RESTORE=RESTAURE
  ";
1350 PRINT @ 520,"FOR=PARA       STOP=CESE        CLS=BORRE
  ";
1360 PRINT @ 584,"ON=EN          STEP=GRADA       GOSUB=VAYA SUB
  ";
1370 PRINT @ 648,"REM=NOTA       LET=HACE         RETURN=RETORNE
  ";
```

```
1380 PRINT @ 714, STRING$(50, 32);
1390 IF INKEY$ ="" THEN 1390
1400 IF FLAG=1 RETURN
1410 CLS
1420 RETURN

1425 ' *** Get Keyboard Input ***

1430 PRINT">";
1440 P1=0
1450 LINE INPUT A$
1460 COM$=LEFT$(A$, 4)

1465 ' *** Check for Command ***

1470 IF COM$="ALIS" THEN 1890
1480 IF COM$="AHOR" THEN 2050
1490 IF COM$="CARG" THEN 2180
1500 IF COM$="LIST" THEN 2330
1510 IF COM$="AYUD" THEN GOSUB 1200: GOTO 1430
1520 IF COM$="HELP" THEN GOSUB 1200: GOTO 1430
1530 IF COM$="CORR" THEN 280
1540 IF COM$="NUEV" THEN 280
```

Fig. 16-3. Program listing for Translator. (Continued from page 191.)

```
1550 IF COM$="BORR" THEN CLS: GOTO 1430
1560 IG$=A$
1570 A$=A$+CHR$(32)
1580 B=INSTR(A$, C1$)
1590 C=INSTR(A$, C2$)
1600 IF C=0 AND B=0 THEN 1680
1610 IF B=0 THEN 1670
1620 W$=MID$(A$, B+1)
1630 P1=INSTR(W$, C1$)+B
1640 IF C<B THEN 1670
1650 IF C>P1 THEN 1670
1660 GOTO 1680
1670 IF C<>0 THEN PRINT"SOLAMENTE UNA DECLARACION CADA LINEA":
GOTO 1430
1680 T$=""

1685 ' *** Check for line number ***

1690 :  FOR T=1 TO LEN(A$)
1700 :    IF MID$(A$, T, 1)=CHR$(32)THEN 1730
1710 :    T$=T$+MID$(A$, T, 1)
1720 :  NEXT T
1730 LI=VAL(T$)
1740 IF LI>L2 PRINT"COMENCE LA LINEA CON UN NUMERO MENOS QUE
```

```
";L2: GOTO 1430
1750 IF LI<1 PRINT"COMENCE LA LINEA CON UN NUMERO": GOTO 1430

1435 ' *** Look for Spanish keywords ***
1760 :  FOR G=1 TO NW
1770 :    P=INSTR(A$, SPAN$(G))
1780 :    IF P>0 THEN 1830
1790 :  NEXT G
1800 E2$(LI)=IG$
1810 CP$(LI)=A$
1820 GOTO 1430
1830 IF P<B THEN 1860
1840 IF P>P1 THEN 1860
1850 GOTO 1790
1860 L=LEN(E$(G))

1865 ' *** Make Substitution ***

1870 MID$(A$, P, L)=E$(G)
1880 GOTO 1790

1885 ' *** List Spanish Program Lines ***
```

Fig. 16-3. Program listing for Translator. (Continued from page 193.)

```
1890 V=INSTR(A$, C3$)
1900 IF V=0 THEN 1960
1910 V2$=MID$(A$, V)
1920 V3=VAL(V2$)
1930 IF V3>0 THEN PRINT E2$(V3) ELSE 1960
1940 PRINT
1950 GOTO 1430
1960 CU=1
1970 CLS
1980 :  FOR N=1 TO 200
1990 :    IF E2$(N)="" OR E2$(N)="," THEN 2030
2000 :    PRINT E2$(N)
2010 :    CU=CU+1
2020 :    IF CU/14=INT(CU/14)THEN PRINT"EMPUJE < ENTER >";:
INPUT E$
2030 :  NEXT N
2040 GOTO 1430

2045 ' *** Save Programs to Disk ***

2050 INPUT"NOMBRE DE LA PROGRAMA EN ESPANOL :";NE$
2060 INPUT"NOMBRE DE LA PROGRAMA EN INGLES :";NI$
2070 OPEN"O",1, NE$
2080 :  FOR N=1 TO 200
```

```
2090 :    PRINT#1, E2$(N); CHR$(13);
2100 :  NEXT N
2110 CLOSE 1
2120 OPEN"O",1, NI$
2130 :  FOR N=1 TO 200
2140 :    PRINT#1, CP$(N); CHR$(13);
2150 :  NEXT N
2160 CLOSE 1
2170 GOTO 1430

2175 ' *** Load Programs From Disk ***

2180 INPUT"NOMBRE DE LA PROGRAMA EN  ESPANOL :";F3$
2190 F$=LEFT$(F3$, 8)
2200 INPUT"NOMBRE DE LA  PROGRAMA  EN INGLES :";F3$
2210 F3$=LEFT$(F3$, 8)
2220 OPEN"I",1, F$
2230 :  FOR N=1 TO 200
2240 :    LINE INPUT#1, E2$(N)
2250 :  NEXT N
2260 CLOSE 1
2270 OPEN"I",1, F3$
```

Fig. 16-3. Program listing for Translator. (Continued from page 195.)

```
2280 :  FOR N=1 TO 200
2290 :    LINE INPUT#1, CP$(N)
2300 :  NEXT N
2310 CLOSE 1
2320 GOTO 1430
2330 CU=1

2335 ' *** List Programs ***

2340 :  FOR N=1 TO 200
2350 :    IF CP$(N)<>"" THEN PRINT CP$(N): CU=CU+1
2360 :    IF CU/14=INT(CU/14)THEN PRINT"EMPUJE < ENTER >";:
INPUT E$
2370 :  NEXT N
2380 PRINT
2390 GOTO 1430
```

Fig. 16-3. Program listing for Translator. (Continued from page 197.)

the loop which searches for keywords got to SIGUIENTE, the word would have been changed to IFGUIENTE.

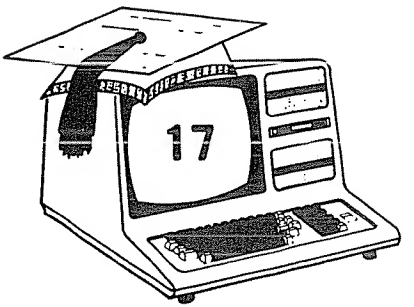
Examining the program, described in Figs. 16-2 and 16-3, we can see that actual translation from Spanish to English is simple. The programmer enters a line, which is loaded into A$ in line 1450. The first four characters are checked to see if any of the allowable commands are included. If not, then the line must begin with a line number, or an error message will be generated. A check is made for a colon outside quotes, which would indicate a multiple-statement line. An error trap also checks to make sure that the line number is within the range allowed.

A FOR-NEXT loop beginning at line 1760 compares each word in the line with the permissible keywords, and, if one is found, the substitution for the equivalent English keyword is made. Several subroutines take care of LISTing the program lines, stored in two string arrays.

The only hitch in Translator is a problem common to all compilers. The programmer cannot run the program to test it until it has been compiled. Then, if bugs are found, the compiled version cannot be changed (because, in this case, the Spanish-speaking person supposedly cannot understand the BASIC object code). Of course, an English-speaking person can edit it, but for those for whom Translator was intended the object code may mean about as much as a machine-language dump.

Because Translator was meant as a learning tool, it was designed to be easy to change. Keywords can be added by appending them to the proper locations in the DATA lines, adding numeric DATA that shows the difference in length between the longer Spanish keyword and the shorter English equivalent. WR must also be changed to reflect the new number of words.

This program will compile from any language. The user could select keywords in, say, French, and enter them with their English BASIC counterparts in the DATA lines. All the prompts in Spanish will have to be changed as well, but these have purposely been kept to a minimum in the program.



Document Sorter

The final program in the book will be of use to anyone who writes or uses long documents, or who is interested in the clarity of their writing style—even if they generate nothing longer than a memo or letter. Document Sorter will provide you with an alphabetized list of every word in a document, or even a book-length manuscript. In fact, the program was written for that very purpose.

Are you preparing an index or glossary for a term paper, article, or book you are working on? Curious about the scope of your vocabulary? Document Sorter will take most text documents of reasonable size, throw out the punctuation marks and numbers, and collect the remaining words into a file sorted alphabetically. Duplicates and many plurals of a root word are also ignored, so that you wind up with a listing only of the unique words in your document.

This program was written recently to help in the preparation of a particularly technical book. After about 60,000 words were run through it, the result was a list of a few thousand unique words that was further condensed to form a glossary and index. Document Sorter will also work with your shorter text items, such as letters, short stories, or school assignments. Odd punctuation won't throw it, and capitalized words are automatically converted to lowercase. You can even use the program on your BASIC programs to find out what keywords were used. Line numbers and other non-alphabetic characters will be thrown out as well.

Those who want to measure the "fog index" in their writing can

use the program to compile a list of the commonly used words in their written vocabulary. Just append a representative number of letters or memos together, or any other documents that you want to study, and run the program. Since duplicate words are eliminated, the listing produced is a fair gauge of a vocabulary. If you wish, changing a single line in the program will keep Document Sorter from dropping the duplicates; in this case actual frequency of word usage can be measured.

Note that you will need a text processor to generate the text files to be checked. The program was tested with Scripsit files, which were saved in ASCII form using the "S,A filename" syntax. The program will work with any ASCII file, including programs, if the goal is to alphabetize the words used in the program listings. Line numbers, other numeric constants, and many single-character variables will be filtered out as well. Variable names with two or more characters will be left alone, and sorted along with keywords and words within prompts.

Also note that the program does not incorporate a sorting routine of its own. Instead, it simply acts as a sophisticated filter, and relies on the fast machine-language sort built into many operating systems. The sort used here is the CMD"O" sort found in NEWDOS/80 2.0. This works with TRS-80 Models I and III, or Model 4 operating in Model III mode.

Model III TRSDOS also has a similar CMD"O" string sort, and some other operating systems may have their own provisions for fast, machine-language sorts. Various utility programs can also be adapted. Because of the size of the arrays used in this program (up to 4,000 elements), I didn't bother to include a BASIC sort routine, which would be much too slow. Since Document Sorter requires a disk drive anyway, most users should have an operating system with a built-in sort.

Several interesting programming tricks make Document Sorter an educational as well as useful module. Warning: This program uses a *lot* of string space. As a result, it can take quite a while to sort a fairly long document, say, one 3,000 to 4,000 words long (that's 13 to 18 double-spaced typewritten pages). The time needed is the result of string "garbage collection," during which process the computer may appear as if it has locked up.

While there might be more efficient ways of carrying out the designed functions, ones that would not involve this string collection bottleneck, the most obvious method was chosen for this book because it was quicker to write, and easier to explain.

Your entire document to be sorted is loaded into memory at one time, and is stored in a string array, WRD$(n). This array is DIMensioned to 4,000 elements in line 80, and allows a document with a maximum size of 4,000 words. This will handle most documents, although I had to break my book up into about 15 different sections. Once the program had thrown out all the duplicates and nonwords in each section, I was left with a much shorter collection of words. These were merged and run through the program additional times until I ended up with a single sorted file.

Although the machine-language sort of your array is very fast, parsing the document into individual words takes time. I'd recommend letting the program run while you do something else. Shorter documents can be processed much faster than the equivalent amount of text in a single, longer document. That is, four 1,000-word files will be sorted faster than one 4,000-word document.

The reason for this is that, as mentioned, some time is taken up in so-called "garbage collection" as string space is consumed by your program. With longer documents, the available string space becomes smaller as more of the document is processed, so string collection must be done more frequently.

Shorter documents do not require so much time at this task, and thus run to completion sooner. Since I had time to spare, I wrote the program to allow maximum document size, even if the longer documents frequently slowed down the program. While it might take two or three minutes to process a 1,000-word file, it can take 10 minutes (or longer) to handle a 4,000-word document. You can let your computer operate unattended or, as I did, allow it to process many files consecutively overnight. The way to do this will be described later.

Garbage collection of unused string space can also be cut down by CLEARing as much space at the beginning of the program as possible. With a 4,000-element array, only about 24,000 bytes were free for strings (line 60). If you will be sorting only shorter documents, you can decrease the size of the array to, say, 2,000 elements, and increase the available string space to 30,000 bytes or more.

To help speed up Document Sorter, variables were defined as integers (line 90). In addition, Model 4 owners can accelerate their computer's operation by including line 70. This OUT statement allows running the Model 4 at full 4 MHz speed, even in Model III mode. Owners of "real" Model III or Model I computers will want to delete line 70.

The first step in setting up Document Sorter is to decide what characters will be considered word "delimiters." That is, how do we know when we have reached the end of a word? Obviously, a space (CHR$(32)) indicates the end of a word. But most punctuation, such as commas, periods, question marks, and (usually) apostrophes, also marks ends of words. Hyphens and other characters such as parentheses and asterisks are also not found within words. So Document Sorter looks for any of these and, if one is found, ends the current word. The only exception is when a character *other than* "s" follows an apostrophe. Thus, "can't" or "I've" would not be truncated after the apostrophe, but the terminal "s" on "America's" would be cut off. This effectively filters out possessive forms of words, while allowing most contractions. A few contractions end in "'s" (such as "it's" and "let's"), but those are minor exceptions that don't detract seriously from the usefulness of this program.

Fortunately from a programming standpoint, most of the word delimiters used by Document Sorter occur all in one place in the standard ASCII list—CHR$(21) to CHR$(64). The program builds one long string, containing one example of each, in lines 100 to 120. At this point, DELIMIT$ is equal to !"#%–'()*,-./0123456789:;<=>?@. Three more characters, CHR$ (140)-CHR$(142), are used by Scriptsit to mark the ends of lines, paragraphs, and pages. These are added to the list so that Document Sorter will recognize them as end-of-word markers and filter them out. If your own word processing program uses special CHR$ codes, add them to DELIMIT$ in line 130.

The next section, lines 140 to 200, asks the user for the desired filename to process, and the filename of the output file (the sorted words). If you have many documents to sort, as I did, you can replace the LINEINPUT statements with a large FOR-NEXT loop beginning at line 180 and ending at line 530. I called each of the files I wanted to sort "CH1," "CH2," "CH3," etc., and stored them on disk under those names. Then, each time through the FOR-NEXT loop, the filenames were constructed automatically:

```
160  FOR CHAPTER=1 TO 10
170  F$="CH"+MID$(STR$(CHAPTER),2)
180  F2$=F$+"/SRT"
. . .
530  NEXT CHAPTER
```

Actual processing of the file begins at lines 210-220, where a

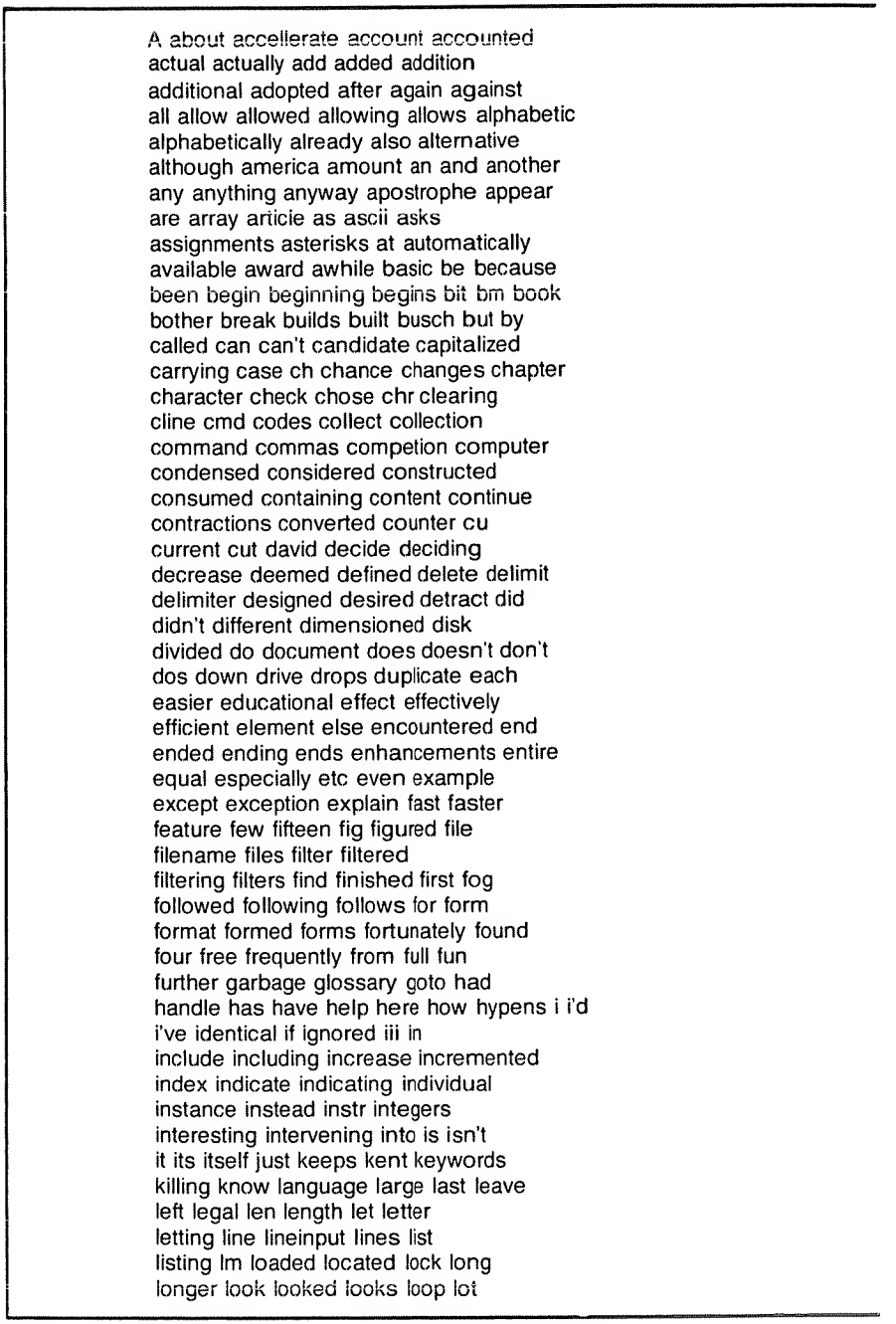


Fig. 17-1. Words in Chapter 17, compiled by Document Sorter.

lowercase ls machine make many mark
marker marks maximum may mhz
memory merged message mid might
minor misnomer mode model module more
most much my name named names
nest newdos next ni non nonwords not nothing
noting now nulled number obvious
obviously occur odd of off offered ohio
on once one only operating operation
operator or other out output own page
paper paragraphs parentheses parsing
particular past periods pf pl place
plural point position possessive
possible preparation preparing present
previous printed process processed
processing program programming programs
proportionately provisions punctuation
question quicker quickly quite ran
range rd reached read real really
reason reasonably recently recognize
recommend reduced remaining reorder
replace requirements requires resisted
result resume right rm root routine run
running s' same save saved saver say
school scope screen scripsit second
section see seriously serves setting
several short shorter shown similar
since single size sized slip slowed smaller so
some something sooner sort sorted
sorter sorting sorts space spare
special speed spend srt standard
standpoint starts statement step stored
stories str string strip successive
such syntax system take taken takes
taking task temptation term terminal
text than that the their them then
there these things this those thousand
three through throw thrown thus time tm
to too total track tricks trs truncated
under understand unique until unused
unwanted up uppercase use used useful
user uses using usually utility valid
value variables variations various very
vocabulary wait want wanted warning was
way we well were what when where which
while whole will wind with within won't
word work working works worth would wrd
write written wrote ws year yet you
your zero

Total words in file : 1885.
Unique words in file ; 538.

```
10 '
20 ' ********************
30 ' *                  *
40 ' *  Document Sorter *
50 ' *                  *
   ********************
60 CLEAR 24000
80 DIM WRD$(4000)
90 DEFINT A-Z

95 ' *** Construct word delimiter string ***

100 FOR N=21 TO 64
110 DELIMIT$=DELIMIT$+CHR$(N)
120 NEXT N
130 DELIMIT$=DELIMIT$+CHR$(140)+CHR$(141)+CHR$(142)

135 ' *** Enter filenames, open files ***

140 CLS
150 PRINT TAB(10)"Document Sorter"
160 PRINT:PRINT
170 LINEINPUT"ENTER FILENAME TO SORT :  ";F$
```

```
180 LINEINPUT"ENTER OUTPUT FILENAME :  ";F2$
190 OPEN "I",1,F$
200 OPEN "O",2,F2$

205 ' *** Read Line From Disk ***

210 IF EOF(1) GOTO 380
220 LINEINPUT#1,A$

225 ' *** Convert to lowercase, check for delimiters ***

230 FOR N=1 TO LEN(A$)
240 B$=MID$(A$,N,1)
250 IF B$=>"A" AND B$=<"Z" THEN B$=CHR$(ASC(B$)+32)
260 PLACE=INSTR(DELIMIT$,B$):IF PLACE<>0 GOTO 270
    ELSE GOTO 350
270 IF B$<>"'" GOTO 300
280 IF MID$(A$,N+1,1)="s" GOTO 300
290 GOTO 350

295 ' *** Add word to array ***

300 CU=CU+1:WRD$(CU)=C$:C$="":PRINT WRD$(CU)
```

Fig. 17-2. Program listing for Document Sorter.

```
305 ' *** Filter out null words, numbers ***

310 IF WRD$(CU)="" OR WRD$(CU)="s" OR VAL(WRD$(CU))>0
    THEN CU=CU-1:GOTO 360
320 IF LEN(WRD$(CU))>1 GOTO 360
330 IF WRD$(CU)="a" OR WRD$(CU)="i" GOTO 360 ELSE GOTO 340
340 CU=CU-1:GOTO 360

345 ' *** Add character to current word ***

350 C$=C$+B$
360 NEXT N
370 GOTO 210

375 ' *** Sort array ***

380 CLS
390 PRINT"SORTING FILE OF ";CU;" ITEMS."
400 CMD"O",CU,WRD$(1)

405 ' *** Write non-repeating words to file ***

410 FOR N=1 TO CU
```

```
420 IF WRD$(N)=WRD$(N-1) GOTO 480
430 IF WRD$(N-1)<>LEFT$(WRD$(N),LEN(WRD$(N-1))) GOTO 450
440 IF RIGHT$(WRD$(N),1)="s" THEN GOTO 480
450 PRINT WRD$(N)
460 PRINT#2,WRD$(N);" ";
470 NI=NI+1
480 NEXT N

485 ' *** Write results to file ***

490 PRINT #2,""
500 PRINT#2,"Total words in file : ";STR$(CU);"."
510 PRINT#2,"Unique words in file :";STR$(NI);"."
520 CLOSE

525 ' *** RUN again? ***

530 CLS:PRINT:PRINT
540 PRINT "Do you want to sort another file?  (Y/N)"
550 A$=INKEY$:IF A$="" GOTO 550
560 IF A$="y" OR A$="Y" THEN RUN
570 CLS
```

Fig. 17-2. Program listing for Document Sorter. (Continued from page 207.)

line is read in from your ASCII text file. In addition to text files, Document Sorter will also process programs that you have saved in ASCII form (SAVE"filename",A), although most of the content (such as line numbers) will be ignored.

If the end-of-file marker is not encountered, the program starts a FOR-NEXT loop from 1 to the length of the current text line, A$. Each successive character in A$ is looked at, using MID$, and stored in B$. If B$ is an alphabetic character in the range from uppercase A to uppercase Z, it is converted to lowercase in line 250. Then, a programming time-saver is used to see if B$ equals any of the word delimiters.

One way to do this would be to nest yet another FOR-NEXT loop, and check B$ against each delimiter character, which we might have stored in a string array:

```
260  FOR N1=1 TO 46
261  IF B$=DELIMIT$(N1) GOTO 270
262  NEXT N1
```

This would slow things down quite a bit. An alternative would be to use the DELIMIT$ we already defined, and check each character against B$:

```
260  FOR N1=1 to LEN(DELIMIT$)
261  IF B$=MID$(DELIMIT$,N1,1) GOTO 270
262  NEXT N1
```

This is also a bit slow. Instead, INSTR looks at the whole DELIMIT$ string and finds B$, if present, much more quickly. If the position of any of the delimiters does not equal zero, the program drops down to line 270, where it checks to see if B$ equals an apostrophe (line 270). If it does, and the apostrophe is followed by "s", the word is allowed to continue to the next word delimiter. (This might be a space; Document Sorter also does not filter out possessives ending in "s ' ".)

Once a valid word ending has been located, the word counter CU is incremented by 1, and the word, C$, is stored in WRD$(CU). C$ is nulled to begin the next word, and the word is printed to the screen.

Some words are filtered out at this point. If WRD$(CU) equals nothing (" "), or is only the letter "s," or has a value (indicating that it is a number), then CU is reduced to its previous value, effectively killing the word. Single character words other than "a" and "I" are also left out of the array.

When the end of file is reached, the array is sorted. NEWDOS/80 2.0's syntax is shown in line 400; you may have to make some changes to account for your particular DOS or sorting utility. Then most of the sorted words are written to a disk file, with one last filtering taking place.

Beginning at line 410, a FOR-NEXT loop from 1 to CU first checks to see if WRD$(N) equals WRD$(N-1). This would indicate a duplicate word. Only the first instance of a word is written to the disk file. The program also checks to see if a word and the previous word are identical except for a terminal "s." If so, the second word (the plural) is not written to the file. Some plurals *will* slip past, especially those divided from their root word by an intervening word, such as "name named names." I deemed it not worth the bother to look for all plurals, because there are many not formed using "s" that slip into the file anyway.

A counter, NI, keeps track of how many words are actually written to the file. When finished, the program writes a message at the end of the file noting how many total words were in the file, and how many unique words have been printed to the file.

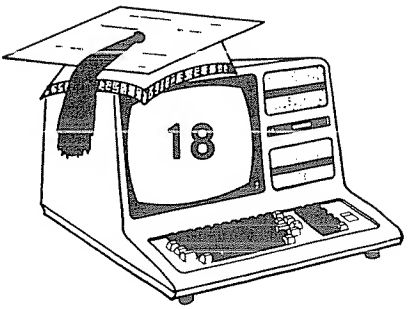
Then the operator is offered the chance to run the program again. Now that you understand the workings of Document Sorter, you might want to add enhancements that will filter out some of the exceptions not accounted for. I've resisted the temptation to include *all* the variations, to leave some of the fun up to you. For example, to look for possessives ending in "s', " just add the following line:

```
415  IF RIGHT$(WRD$(N),2)="s' " GOTO 480
```

or

```
415  IF RIGHT$(WRD$(N),2)="s' " THEN WRD$(N)=
     LEFT$(WRD$(N),LEN(WRD$(N)-1)
```

A sorted list of the words in this chapter, processed by Document Sorter is shown in Fig. 17-1, while the program itself is listed in Fig. 17-2.



Some Tips

The whole aim of this book has been to show you how to make your programming more efficient by letting other programs write your code for you. The 17 programs presented so far generate program lines, modify software, or perform other tasks for you. However, there is no reason to limit your automatic TRS-80 to just those utilities included here. There are actually many, many programs on the market that will streamline your work.

There is one tool you may not have thought of—unless you are an old-time programmer, or write in assembly language or for compilers. That utility is your word processor. Word processors of today have much in common with text editors, used in the past to write programs that are compiled or assembled into machine-language code at runtime. However, most BASIC programmers today have never written a program with a text processor. The majority have worked only with interpreters. An interpreter is, of course, a computer program that takes the instructions written by the programmer, and translates it into the computer's machine language each time a line is run.

That is, when a line like FOR N=1 TO 50:B=A+C:NEXT N is encountered, the interpreter will calculate the machine code fifty different times. This is why interpreters are so much slower than machine-language programs. However, there are advantages to interpreters. One is that a program can be written a small part at a time, with each section run, tested, and then modified immediately.

Another advantage is that interpreters can include error-trapping features that handle user input—such as attempts to store numbers larger than 32,767 in an integer variable—that might have been unanticipated when the program was written.

Compilers and assemblers are less forgiving. Code is written, and the source code used to produce the runtime object code. Mistakes can only be corrected by modifying the source code and then compiling or assembling new object code. Partially because of this, BASIC interpreters have been the favored tool, but TRS-80 BASIC programmers have missed some of the editing and program writing tools possible with word processors.

Unless special utility programs are used, the TRS-80 Models I/III and 4 rely on *line-oriented* editing. That is, if a change must be made in line 40, the programmer types in EDIT 40 and makes changes only to that one line. If a similar change must be made to line 50 (such as changing all the PRINTs to LPRINTs), it is necessary to EDIT that line, and so on throughout the program. Making a lot of changes to an existing program can be tedious and time-consuming.

But wait. What if the program were loaded into the word processor as if it were a document? Then the arrow keys could be used to zip the cursor around the program, and changes could be made by overtyping, global search and replace, and other powerful features.

In truth, this *screen-oriented* program editing is nothing new. Some computer systems, such as the Commodore PET, VIC-20, Commodore 64, and even Radio Shack computers like the Model 100, use this type of editing, or some variation. With the Commodore line, for example, any program line that appears on the screen can be edited simply by placing the cursor over a character and typing, inserting, or deleting as desired. If changes are to be permanent, the programmer hits enter while the cursor is still on the program line. Otherwise, (or by hitting shift RETURN) the changes are ignored. Duplicating a program line can be accomplished simply by editing the line number. A copy of the line appears under the new line number, while the old program line remains.

With the TRS-80 Model 100, entering an edit command for a line, or range of lines, actually causes the portable computer to enter its TEXT mode, with each of the lines specified translated to ASCII form for editing. In this case, however, a program line number can be edited, but the old program line is deleted.

Although utilities are available to give other TRS-80s screen editing, you probably own one already: your word processor. Programs can easily be loaded into Scripsit, edited, and put back on disk. All of the programs in this book, in fact, were at least partially edited using Scripsit.

The only "trick" to using a word processing program as a program editor is to remember to save the program from BASIC in ASCII form. Then it will be loadable into the word processing program. You must also take care to store the program from the WP software in ASCII as well. With Scripsit, the syntax is "S,A filename/ext." If you forget this step and attempt to load the program, only a few characters of garbage will appear on the screen. Don't panic. Return to the word processing program, reload the compressed program file, and then reSAVE it in ASCII.

What can you do with a program in text form? For starters, how about formatted listings even slicker than those produced by Lister? The latter was provided both as an illustration and for those who do not have a WP program. However, Scripsit was used to print out the listings reproduced in this book. The word processing software divided up the program lines into pages, and printed a header at the top of each page.

By setting the window of the TRS-80s screen to the same width as the paper being used, it was simple to scroll down through the program text to see when lines were too long. In most programs, line breaks were chosen for clarity, and the next part of a line indented. Scripsit was also used to add spacing betwen REMarks and the program lines preceding and following.

Although, unlike interpreters, original code cannot be tested by a WP program, there are advantages that make them very desirable. Here are a few tips for using Scripsit to streamline your program writing. Those of you with other WP programs can use them as well, by applying the particular syntax and commands of your favored text processor.

1) Put your most-used modules at the tips of your fingers. Several phrases and program lines were written and encased in blocks given unique markers. Then, when a phrase like A$ = INKEY$:IF A$="" GOTO was needed, it was a simple matter to type"@S@Q," which is the Scripsit command for INSERT BLOCK. When asked the name of the block, the letter for the desired phrase was entered. The inserted block has no block markers and the original block remains available for insertion in other positions.

Of course, it would have been simpler to write subroutines and call these, rather than write the code over and over, even automatically. But "easier" is not always as clear for someone attempting to understand a BASIC program. So, in many cases subroutines were avoided. Programming speed did not slow down, however, because of the power of the word processing program.

2) Global searches, replaces, and deletions made writing the programs in this book much easier as well. Halfway through a program, on discovering that a variable name was ill-chosen, it was a simple matter to replace all occurrences in a few seconds. Or, "REM ***" could be changed to "' ***" almost instantly. Some program screens, written using Screen Editor, had PRINTTAB(0) in a number of places. All the "TAB(0)" appearances could be deleted just as quickly.

Care has to be taken when using this feature, however. A word processor will not check to see if the string being changed is inside quotes or not. Changing all PRINTs to LPRINTs can result in some undesired modifications, such as LPRINT becoming LLPRINT, or "IS YOUR PRINTER ON?" transformed into "IS YOUR LPRINTER ON?"

3) Programs can be "cleaned up" quite easily. It is fast and efficient to zip through a program with a word processor and touch up sloppy coding, change all uppercase prompts to upper and lowercase, or delete undesired spaces. After writing Tabber, I wanted to go through some earlier programs and center prompts. However, some program lines had prompts with (horrors!) embedded spaces:

```
10  PRINT " DO YOU WANT TO:"
20  PRINT "       1) RUN A PROGRAM"
30  PRINT "       2) EXIT THIS PROGRAM"
40  PRINT "          ENTER CHOICE:"
```

While it was easy to type embedded spaces when writing the original program, someone typing in the program from this book would be hard-pressed to count the number of spaces needed to properly format the lines on the screen. By replacing all PRINT " with 'PRINT" one space was closed up. Then, by replacing all 'PRINT" 'with simple PRINT and quotation marks, the excess spaces inside the prompts were elminated. The PRINTAB(T)s could then be put where needed, and Tabber did the rest.

Utilities need not stop at word processors, either. How much

would you pay for a program that would allow you to insert your 10 or 11 most-used subroutines into a program you are writing, at the touch of a key? Would you like to be able to format a disk or produce a backup copy by touching a key, or hit another to call up your word processor, another utility, or BASIC?

There are a number of machine-language programs available that will let you do just that. Many communications programs, SuperScripsit, and other utilities also allow you to define keys. The TRS-80 Model 100 supports redefining the special function keys (but only up to 15 characters), while the TRSDOS 6.0 supplied with the TRS-80 Model 4 has its own keystroke multiplier capabilities.

Those who think that special function keys are best applied as a kind of shorthand to eliminate typing in GOTO or other phrases suffer from a failure of imagination. Programs which allow users to define keys—a program called IRV, for example, allows keys to be defined up to 255 characters—can be very useful in other ways.

The nice thing about general-purpose microcomputers is that they can be custom-configured to perform specialized tasks tailored to the exact needs of the end user. Thanks to sophisticated disk operating systems, patches, special ROMs, and utility programs, many features can be available on power-up or, at most, at the press of a few keys. User-programmable special function keys can do a great deal more than printing out a lengthy BASIC keyword.

Programmable function keys vary in their utility. Some programs allow reprogramming every key on the keyboard, while other systems limit you to only one or two special function keys. You may have tight limits on the number of keystrokes that can be programmed.

Many DOS functions can be compressed into several special function keys. User-programmed keys also allow duplicating (and improving on) other advanced DOS features with lesser operating systems. It is possible to change the default drive number for DIR under many types of DOS, so that every time you enter DIR you see the directory of, say, Drive 1, instead of that of the system disk on Drive 0. Program a key so that a shift key produces "DIR :1." Other keys can be programmed to provide directories of other drives—by hitting two keys instead of seven.

Striking <shift>!on a TRS-80 can yield a COPY command, complete with all the necessary <ENTER>s, to make a full disk copy from Drive 1 to Drive 2. <Shift>X can produce "BOOT," allowing you to reboot the system without hitting the reset key or typing the full word.

You can store several keyboard configurations with many programs. Have one setup for programming, another for operating. While three or four different configurations might be possible, it is unlikely that an operator would bother to learn all those commands, or mark the key fronts with all possible combinations.

Because some special function key programs allow multiple-line programming, there is no reason why you can't define keys with useful subroutines, which can then be added to your program at a single stroke. For example, you can build the routine shown in Fig. 18-1 into a key program.

These two subroutines, because they total less than 255 characters, can be loaded into a single programmable key with a program such as IRV. Then, when writing code that requires an all-purpose disk I/O routine, simply hit <shift>W and the above lines will appear. By keeping the main program line numbers less than 10000, the appending is automatic. If necessary, the standard subroutines can be edited, renumbered, or moved as required. Subroutines aren't the only programs that can be entered as key definitions. Any short program can be stored, such as your gas-mileage calculator, a binary/decimal/hex converter, or a program that prints out a frequently used expense report.

Or, try this program for one key:

```
10000  A$=INKEY$:IF A$=" " GOTO 10000
10100  INPUT A:B=B+A:PRINT B:GOTO 10100
10200  FOR N=32 TO 191:PRINT N;" .)";CHR$(N);"
       " ";:NEXT N:STOP
```

```
10000  OPEN "O",1,F$
10100  PRINT #1,NI
10200  FOR N =1 TO NI
10330  PRINT #1,DA$(N);","
10400  NEXT N
10500  CLOSE 1
10600  RETURN
10700  OPEN "I",1,F$
10800  INPUT #1,NI
10900  FOR N=1 TO NI
11000  INPUT #1,DA$(N)
11100  NEXT N
11200  CLOSE 1
11300  RETURN
```

Fig. 18-1. Example of function-key subroutines.

By hitting the appropriate key, you can call up three or more short programs that can be accessed by typing RUN 10100, etc. Line 10000 is an input line that can be included anywhere in your program. It serves as an example of typical much-used lines that can be drawn from a stock library. Line 10100 simulates an adding machine whenever a series of quick calculations are needed and you require subtotals, or you don't want to use your computer's command mode. Line 10200 provides a quick listing of alphanumerics and graphics, along with their applicable CHR$ codes for ready access.

User-programmable keys are truly the programmer's friend. Do you frequently renumber your programs during writing to make additional room between lines? Program a key to yield "RENUM 10,10 <ENTER>" (or whatever syntax your BASIC uses) whenever you strike it. Another key could produce a REF variable cross reference listing.

Your uses are limited only by the number of keys available for programming. Having a key produce CMD"DIR" would be the equivalent of the F1 (FILES) special function key on the Model 100. You can come up with other applications not suggested here.

As a program is developed, it is good practice to save the work in progress periodically, either to cassette or disk. Should a power failure occur, hours of work are not lost. With disk-based systems, backups are much easier—so simple, in fact, that many programmers end a session, look at the working disk's directory, and see 10 or more versions of a program tucked away. This system works fine, but few of us can remember what we called the last version. Either we invoke a SYSTEM or CMD "DIR" to check, or play it safe and skip a number or two.

Here's a short program for disk users that can be appended to the end of any program you are working on, and used to automatically SAVE an updated version of a program under an appropriate filename. To use it, remember to turn on your CLOCK. Then, by typing GOTO 30000 at any point during program development, the module will collect the current TIME$, extract the hour and minutes, and use that to make the filename.

```
30000  B$=TIME$:H$=MID$(B$,10,2)
30010  M$=MID$(B$,13,2)
30020  F$="PROG"+H$+M$
30030  SAVE F$
```

Save this in ASCII form on your disk, and then APPEND or MERGE it to any program you choose (as long as it does not have line numbers which conflict). You may want to EDIT line 30020, replacing the string "PROG" with any four letters that are more meaningful for the program you are developing. If you want to back up the program to two (or more) disk drives automatically, add the following lines:

```
30025  F1$=F$+" :1":F2$=F$+" :2"
30035  SAVE F1$:SAVE F2$
```

These two short examples are just two of the utilities you can write yourself to make your programming easier. This book should have given you ideas for many others. The goal of the automatic TRS-80 is to let the computer do all the work, and the programmer do all the creating.

Appendix

Converting Model III Programs to the Model 4

One of the strongest advantages of purchasing a TRS-80 has been the high degree of compatibility of all the computers in the line, at least in the area of BASIC programming. If one avoids exotic file types, such as those added by NEWDOS/80, and obscure commands that aren't needed for most applications, the same Disk BASIC programs written for the TRS-80 Model I in the late 1970s can be used with the Model 4 of the mid-1980s. All such programs can be used on the Model 4 in Model III mode; many will be compatible in Model 4 configuration as well.

This is also true for the programs in this book. They will run as-is on Model I/III computers, or the Model 4 emulating a Model III. All but a few can easily be converted to operate under TRSDOS 6.0 in full-fledged Model 4 mode. Here are a few tips on making the conversion.

Model III and Model 4 BASICs are very similar in many respects. However, there is a difference in the way the two computers handle video displays. Models I and III both store information about what is displayed on the screen in 1,024 memory locations, beginning at 15360. Two of the programs, Screen Editor and Visual Maker, prepare program lines based on what has been written to the screen by the user. These two cannot easily be transferred over to the Model 4 mode. However, both can be used with Model 4 computers in Model III mode, and the resulting programs transferred to Model 4 TRSDOS disks.

Two other programs, Error Trapper and Chain Zapper, are fairly specific for the disk operating systems of Models I and III. All the other programs can and have been transferred to a Model 4 computer, and with only a few changes, successfully run.

Because of the scarcity of Model 4 software, most purchasers of the computer will also have access to a Model III operating system such as TRSDOS 1.3, NEWDOS/80, LDOS, DOSPLUS, MULTIDOS, or a compatible system. Others might have a Model I computer, with the Model 4 as a second, or replacement unit.

Such users might find it convenient to enter the programs in the alternate mode, so they can be tested and compared with the program listings in the book before conversion. Having a Model III operating system will also allow full use of the two programs which do *not* convert easily.

Once you have the programs on a Model I or Model III disk, it is necessary to save them in ASCII form. Then load the TRSDOS 6.0 disk. If only the two built-in drives are available, it may be most convenient to KILL extra files on the TRSDOS disk, to leave room to transfer programs. With a Model 1 disk in Drive 1, simply type REPAIR 1. This will make the necessary changes to the single density Model 1 disk so that TRSDOS 6.0 can read it. Copy all the files from the Model I disk to the TRSDOS 6.0 disk.

If the programs are on a TRSDOS 1.3 disk, the correct command is CONV :1 :0 (VIS,Q=N). Because NEWDOS/80 allows changing the disk directory starting "lump," the directories of NEWDOS/80 Model I and Model III disks can differ from those of TRSDOS 1.3. The PDRIVE specifications can vary widely as well. The most foolproof way I have found when using NEWDOS/80 is simply to put the programs on a single density Model I formatted disk, and load into the Model 4 using REPAIR. For both Model I and Model III, the NEWDOS/80 PDRIVE should be set to TI=A TD=A.

Once the program is deposited on a Model 4 disk, I recommend going to BASIC and loading "GLOBAL" first of all. Check through the program to make sure that there is a space between "PRINT" and "TAB," and "LINE" and "INPUT." That, and the necessity to make sure that THEN is always included in IF statements, are the only changes necessary to all the rest of the programs. By checking over Global Replacer first, you can use it to process all of the other programs automatically. We told you that the TRS-80 would do all of the work.

Next, do as outlined above. Run Global Replacer on each of the

other programs. Because Model 4 BASIC allows long variable names, the spaces between variables and keywords are *not* optional. If you happened to leave out spaces while typing in programs from the book, the programs will test OK in Model III mode, but will not operate under the Model 4 operating system. Global Replacer can SEARCH for each occurrence of PRINTTAB and replace it with PRINT TAB. It will do the same with LINEINPUT and LINE INPUT. Then run the programs to see how they work. In most cases, all will be fairly well.

Small syntax errors may appear. In Model I/III BASIC, a statement like "IF N=2 PRINT "N=2" is perfectly legal. Many of us who have been using these computers for long periods will leave out the THEN, even if it appears in a program listing. The Model 4 requires THEN. If a Syntax Error appears in a line containing IF, look to make sure that THEN is in the proper place.

Another typical problem will be in long variable names. Under Model I/III BASIC, only the first two characters are significant. The Model 4 allows up to 40 characters. Those accustomed to the earlier Radio Shack computers sometimes get sloppy and abbreviate variable names to only two characters in some places, but use longer names in others.

Look at Translator in Chapter 16. The variable array SPAN$(n) is defined, but under the Model I/III mode it can be referred to as SP$(n), as it is. However, the Model 4 will see those as two different arrays. Check your programs carefully to make sure that some of the longer variable names have not been misspelled or changed from place to place in the programs.

CLEAR is used to allocate string space in the TRS-80 Model I/III computers. With the Model 4 such space is allocated dynamically, so this statement clears the value of all variables and closes all OPEN files (like RUN). It can also be used to set the highest memory location for BASIC, and the amount of stack space. Since none of these functions are required for the programs in this book, you can DELETE the CLEAR statements if you wish.

ROW, which is sometimes used as a counter for two dimensional arrays (e.g., ADDRESS$(ROW,COL)) is actually a keyword under Model 4 BASIC. ROW(y) returns the row position of the cursor. A few other words which could safely be used as variables with the Model I/III are implemented as statements in the Model 4. These include SWAP, WAIT, WHILE, WEND, and WRITE. Extra operators, such as MOD, are provided as well. None are used in any of the programs in this book.

Because screen displays are 80 columns in the Model 4, and only 64 columns in the Model I/III, a different arrangement may appear in some programs. Most programs with PRINTTAB(n) will operate just fine, but the tabbing will not center the text on the screen. There will be extra space at the right side. This is no great problem. If you wish, you can go through the programs and substitute PRINTTAB(T) for those prompts you would like centered. Then run Tabber on the target program.

Programs using PRINT@will definitely cause strange screens. PRINT @ works with the Model 4, but the locations are different. You may have to fiddle with the few PRINT@ statements to come up with new locations which look better.

Some changes will have to be made to Menu Master to account for the differences in your operating systems. The program calls various DOS functions, using NEWDOS/80 syntax. Instead of CMD"DIR" you will want to use SYSTEM"DIR" under TRSDOS 6.0. Use BACKUP instead of COPY, DEVICE instead of PDRIVE or SYSTEM, and make other changes as desired.

Index

Teach Your TRS-80™ to Program Itself!

If you are intrigued with the possibilities of the programs included in *Teach Your TRS-80™ to Program Itself* (TAB BOOK No.) 1798, you should definitely consider having the ready-to-run disk containing the software applications. This software is guaranteed free of manufacturer's defects. (If you have any problems, return the disk within 30 days, and we'll send you a new one.) Not only will you save the time and effort of typing the programs, the disk eliminates the possibility of errors that can prevent the programs from functioning. Interested?

Available on 40-track double-density disk for Radio Shack TRS-80 Models III and 4, with 48 K and two disk drives at $19.95 for each disk plus $1.00 each shipping and handling.

I'm interested in the ready-to-run disk for *Teach Your TRS-80™ to Program Itself*. Send me:

_____ disk for TRS-80 Models III, and 4, 48K (6051S)
_____ TAB BOOKS catalog

_____ Check/Money Order enclosed for $_____

plus $1.00 shipping and handling for each disk ordered

_____ VISA _____ MasterCard

Account No. _____ Expires _____

Name _____

Address _____

City _____ State _____ Zip _____

Signature _____

Mail To: **TAB BOOKS INC.**
P.O. Box 40
Blue Ridge Summit, PA 17214

(Pa. Add 6% sales tax. Orders outside U.S. must be prepared with international money orders in U.S. dollars.)

TAB 1798

Teach Your TRS-80™ to Program Itself!

by David Busch

Why spend hours writing program code when your TRS-80 can do it for you? It's true . . . using the 16 programs included in this unique guide, your TRS-80 Model I, III, or 4 can generate ASCII files that can be loaded and run as programs.

Here is everything you need to put your TRS-80 to work composing subroutines, completing programs, setting TABs, or giving existing software new capabilities and power. Each program is fully explained and is readily adaptable to most TRS-80s.

Let "Visual Maker" design a custom "slide" to appear on the screen of your TRS-80, using graphics and alphanumeric characters. You decide how long the "slide" should be displayed, add more "slides" and soon your TRS-80 will write complete BASIC programs to display the "slides" as you requested! Want to center your screen output for prompts and other messages? Tabber can do it for you, quickly and easily.

All the groundwork for producing "user friendly" programs with title screens, menus, and on-screen directions that are centered and polished-looking can be done by the programs in this book. Think of the time you can save using "Proofer" to find your misspelled keywords, mismatched parentheses, and other errors *before* you run a program!

If you're a beginning programmer, "Error Message" can't be beat. It will locate your errors, plus, detail exactly what the problem is and give you tips for tracking down every bug!

If you write programs to sell, this guide will save you countless hours of programming. (In fact, the author used some of these programs to write other programs included in this guide.) Similarly, you can take ideas and suggestions here and develop programs of your own that will streamline your BASIC development work.

The possibilities are virtually unlimited once you put these innovative programs to use. Whether you're a novice or an experienced programmer, this invaluable guide is guaranteed to save you hours of time on every program you write.

David Busch has been involved in the computer industry since 1974 as a reporter, computerist, and author of more than 300 articles and seven books on computer-related topics.

TAB TAB BOOKS Inc.
Blue Ridge Summit, Pa. 17214

FPT > $11.50

ISBN 0-8306-1798-1

PRICES HIGHER IN CANADA

1095-0784